VISUAL KBOX

by

Paul J. Kaiser

First Paperback Edition 2021
ISBN: XXXXXXXXXX

Printed in the United States of America

Please visit my personal website at:
http://www.cs.lewisu.edu/ kaiserpa/

Contents

Preface

This particular text is not really intended to be a stand-alone book on graphical programming.

Rather it is a companion piece to my previous book **Fun With Programming Languages**. That book focused on the basic elements of programming languages.

For example,

Categories of Languages

- low-level languages: machine and assembly
- procedural languages
- functional programming

Types of Translators

- interpreters
- compilers

Implementation Options

- character sets
- building block components
- syntax
- semantics
- aggregation of data
- transfer of formal arguments
- shallow copy versus deep copy
- types of storage: global, automatic, dynamic

Components in a Translator

- lexical scanner
- syntax checker (parser)
- semantics checker (code generator)
- error handling

The book was divided into four parts. Each part was intended to focus on one general topic and to actually implement a useful computer project based on that material.

- Part One focused on assembly language and asked the reader to build an assembly language interpreter for a hypothetical architecture (**kbox**) and its hypothetical assembly language (**kcode**).
- Part Two focused on a very simple procedural programming language and its implementation. The programming language itself was to simulate an arithmetic calculator (**calc**) by building a compiler that would generate **kcode**. The reader was introduced to scanners, parsers, regular expressions, context free grammars, simple error handling, and code generation for a very basic programming language.
- Part Three focused on another hypothetical programming language (**kize**) which, although limited in features, would give the reader exposure to a variety of implementation options to explore. The reader was introduced to many compiler construction techniques culminating in a fully-function compiler from **kize** to **kcode**.
- Part Four focused on a real, not hypothetical, programming language (**scheme**). However, I did rename the interpreted version we implemented **skeme**. I particularly enjoyed the fourth part of the book! My background is in mathematics. LISP very true to its mathematical roots, was one of the first programming languages implemented, and is very close to the lambda calculus which provides the theoretical basis for all computing. It is just fun to play with. And yet so many students and even practitioners in computer science either dislike it or are unfamiliar with it.

When I had finished writing **Fun With Programming Languages** I realized that when I first started seriously working with X86_64 assembly language I had a very excellent resource in a book by Ray Seyfarth. The title of the book was **Introduction to 64 Bit Intel Assembly Language Programming for Linux**, 2012, ISBN 978-1478119203.

What I had especially liked in the book was his **ebe** X86_64 assembly language simulator. I realized that a new student encountering assembly language for the first time really must make a *leap of faith* that what the assembly language manual describes is really taking place. A graphical user interface at the assembly language level is of great education value. The student can more easily grasp some of the stranger instructions; and the student can see the beauty in a sequence of instructions working in unison to achieve a result.

So I asked myself: Myself! Do you think you can do something like an **ebe** for **kbox**? You have never done anything with a graphical user interface.

My decision to attempt to do the project is described in this book. It is as self-contained as possible. It is quite redundant and repetitious if you happen to have seen my previous book.

Read and enjoy.

All my books are available to download from my personal website at:

www.cs.lewisu.edu/∼kaiserpa

Chapter 1

KBOX Revisited

The KBOX Workout Machine!

1.1 What are KBOX and KCODE?

In a previous book, **Fun With Programming Languages**, I introduced a theoretical 64-bit computer (called **KBOX**)together with an idealized assembly language (called **KCODE**).

My rationale for creating these items was **not** to fill some void that persisted within existing assembly languages, such as X86_64 and AARCH64. Rather it was to avoid a few stumbling blocks in such low level languages that are unnecessary impediments to a new student of assembly language programming – varying size data chunks (byte, word, double word, quad word) and stack alignment.

Please understand that I am not saying these items are unimportant or unnecessary topics! They certainly are and they must eventually be considered, but at a later date. Iinitially a new assembly language programmer should focus on data representation, data manipulation, memory organization, the stack, the heap, and the call / return mechanism.

The next section summarizes the basic elements of the KBOX architecture and the KCODE instruction set. Additional exposition may be found in **Fun With Programming Languages**.

1.2 KBOX Overview

KBOX is an idealized 64-bit computer. It falls in the category of being a RISC machine. The arithmetic logic unit is comprised of 32 64-bit registers.

- Sixteen of the registers will be integer registers, denoted I_0, I_1, ... , I_{15}.
- Sixteen of the registers will be floating point registers, denoted F_0, F_1, ... , F_{15}.

The control unit will contain the program counter (PC) together with the status indicators (FLAGS). There are only three flags that we will be concerned with, all pertaining to the most recent prior comparison: EQ will be true if the two values were equal; GT will be true of the first value was strictly larger; LT will be true of the second value was strictly larger. Only one of the three flags may be true at any given time. We will not concern ourselves with other common status flags, such as overflow, underflow, etc.

All access to the control unit information is indirect through assembly language instructions. The programmer has no direct ability to set the status flags.

The memory unit will also be based on 64-bit chunks. Since everything about the KBOX is built around these 64-bit chunks, we will refer to them as **klunks**.

data types

The **kbox** computer supports all the basic data types as found on most computer systems. But it does not provide the most compact storage capabilities as other machines. We do not worry about different storage units: 8-bit bytes, 16-bit words, 32-bit double words, or 64-bit quad words. Modern assembly languages provide both storage and retrieval capabilities for such items. The **kbox** computer focuses on just one unit of storage, the 64-bit **klunk**. Every data type therefore requires a 64-bit representation.

- unsigned integer / bit pattern / pointer is a 64-bit binary representation **klunk**)
- signed integer is is a 64-bit twos-complement binary representation (**int64**)
- signed real number is a 64-bit IEEE floating point binary representation (**flt64**)
- character is an 8-bit ASCII code representation (**chr64**) stored in the initial 8 bits of the 64 bits
- string is a 64-bit address to a C-string (**str64**)

Character and string data types are really only provided in **kbox** as a convenience to a new programmer.

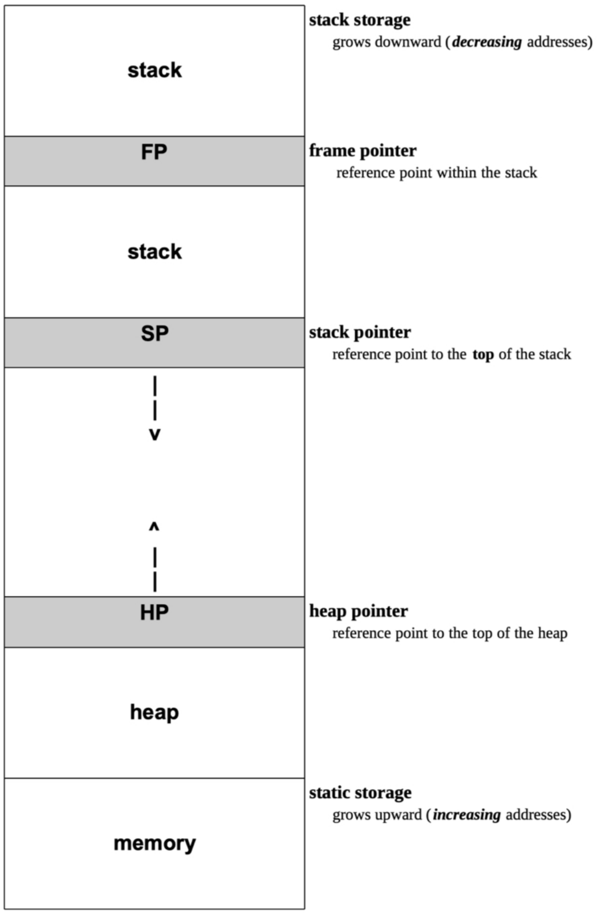
Character data is a horrible waste of data storage; 56 bits out of 64 are empty. But a new programmer can write assembly language code to manipulate arrays of characters without having to immediately delve into **packed** arrays or accessing storage on byte boundaries.

String data has the appearance of being 64-bit data like all the others because actual storage is not in **kbox** memory but in the **visual_kbox** program memory. Strings are only intended to provide an easy way to write input prompts and to assist in descriptive output.

memory

Memory is simply a linear array of 64-bit klunks, each memory cell having a unique address from 0 up to MEMORY_SIZE - 1. Memory cells support the storage and the retrieval of information; all access is done through the memory address.

The following graphic illustrates how **kbox** and most assembly languages tend to organize memory.


stack
stack storage
grows downward (*decreasing* addresses)
FP
frame pointer
reference point within the stack
stack
SP
stack pointer
reference point to the **top** of the stack
HP
heap pointer
reference point to the top of the heap
heap
static storage
grows upward (*increasing* addresses)
memory

It is important to remember that no computation, calculation, or comparison of information actually takes place within memory. These tasks take place within the computer registers.

registers

Assembly languages typically try to anticipate the needs and practices of the programmer. As a result they may pre-assign certain registers for certain purposes; or they may integrate certain helpful instructions into others. **kbox** highlights the following:

The register $\mathbf{I}_0$ may be referred to as **IA** and will be used for arithmetic calculations – typically holding the result.

The register $\mathbf{I}_1$ may be referred to as **IB** and will be used for arithmetic calculations – typically holding the first operand.

The register $\mathbf{I}_2$ may be referred to as **IC** and will be used for arithmetic calculations – typically holding the second operand.

The register $\mathbf{I}_3$ may be referred to as **ID** and may also be used for arithmetic calculations – temporary or alternate values.

The register $\mathbf{I}_4$ may be referred to as **SAR** and will be used as the **source address register**.

The register $\mathbf{I}_5$ may be referred to as **DAR** and will be used as the **destination address register**.

The register $\mathbf{I}_6$ may be referred to as **OR** and will be used as the **offset register**.

The register $\mathbf{I}_7$ may be referred to as **IR** and will be used as the **index register**.

The stack pointer holds the address to the top of the stack; let us reserve register $\mathbf{I}_{12}$ for the stack pointer **SP**.

The frame pointer holds the address to the base of the current activation frame; let us reserve register $\mathbf{I}_{13}$ for the frame pointer **FP**.

The return pointer holds the return address for the current activation frame; let us reserve register $\mathbf{I}_{14}$ as the return pointer **RP**.

The heap pointer holds the address to the heap; let us reserve register $\mathbf{I}_{15}$ for the heap pointer **HP**.

The register $\mathbf{F}_0$ may be referred to as **FA** and will be used for arithmetic calculations – typically holding the result.

The register $\mathbf{F}_1$ may be referred to as **FB** and will be used for arithmetic calculations – typically holding the first operand.

The register $\mathbf{F}_2$ may be referred to as **FC** and will be used for arithmetic calculations – typically holding the second operand.

The register $\mathbf{F}_3$ may be referred to as **FD** and may also be used for arithmetic calculations – temporary or alternate values.

Lastly, the register $\mathbf{I}_{11}$ may be referred to as **ZR** and will be used as the **zero register**. This register was implemented in my original description and implementation of **kcode**; I have chosen to retain it in **visual_kbox** for backward compatability. However, I *strongly* recommend *not* using this feature!

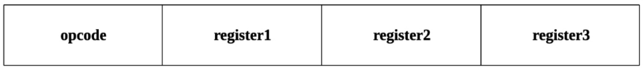
The **zero register** was based on AARCH64 and its implementation of such a register. However, taking away one of sixteen integer registers simply to facilitate moving zero around does not seem terribly useful to me now, especially when the **MOVI** instruction provides even more flexibility in **kcode**.

Note: visual_kbox initializes the zero register to zero. But it makes no attempt to verify that it retains the value zero throughout the duration of any simulation. A programmer may freely use register $\mathbf{I}_{11}$ as an open integer register; any programmer may opt to maintain it as the zero register but must incorporate that decision into his or her coding and must not modify the value in $\mathbf{I}_{11}$.

1.3 KCODE Overview

The idealized KBOX instructions need only be 32-bits! We are not going to be doing anything directly with bit patterns that represent instructions, but we should visualize the typical instruction as having four elements.

The typical instruction format will be an opcode, followed immediately by the destination register for the result, followed by up to two operand registers. Each item occupies one byte in the instruction.



Other configurations for instructions are possible as well, including **immediate instructions** which can include actual data values as part of the instruction. Immediate instructions normally require addition limitations on the value included in the instruction. This is because the data representation bit pattern must appear as part of the instruction bit pattern. Since **kbox** and **kcode** are theoretical and not a real machine, we have the ability to include a 64-bit data value within our 32-bit instruction!

kcode organization

A KCODE program is organized into two segments: a DATA SEGMENT and a CODE SEGMENT.

The DATA SEGMENT is used to define static variables in memory. A DEFINE directive is used to define and initialize variables; a RESERVE directive is used to define storage requirements for variables but with no initialization.

The CODE SEGMENT is used to define the assembly language instructions to be executed. Two important directives should appear at the very beginning of this section: GLOBAL and EXTERN. Subsequent elements must be either a LABEL directive to define a specific location within the executable code for branching purposes or an actual **kcode** instruction.

The two segments may appear in either order; however, each segment must be defined in its entirety and not overlap or interleave elements of both segments.

_DATA_SEGMENT

is comprised of a list of either DEFINE directives or RESERVE directives:

_DEFINE lbl imm$_{64}$
lbl is a variable name
imm$_{64}$ is the initial value
e.g., 123, - 42.95, 'c', "paul kaiser"

_RESERVE lbl size
lbl is a variable name
size is the number of klunks to reserve

_CODE_SEGMENT

is comprised initially of a list of either GLOBAL directives or EXTERN directives:

_GLOBAL lbl
lbl is a name typically identifying where to start
this instruction is totally unnecessary!
the default start is the first instruction

_EXTERN lbl
  lbl is a variable name
    this instruction is also totally unnecessary!
    kbox code can not be linked

following any and all GLOBAL directives and EXTERN directives will be the actual instruction set for the program, intermixed with appropriate LABEL directives:

_LABEL lbl
  lbl is a name to be associated with this instruction
    typically for branching or subprogram activation

The following shorthand notation will be used on the pages that follow to identify and explain the **kcode** instruction set.

- **Ireg** represents an integer register: I_0 , . . . , I_{15}
- **Freg** represents a floating point register: F_0 , . . . , F_{15}
- **reg** represents an arbitrary register of either flavor
- **imm** represents an immediate value
  - integer: 123, -45
  - floating point: -23.56, 145.98
  - hexadecimal: 0X3F
  - character: 'x'   single quotes
  - string: "paul kaiser"   double quotes
  - label: A   no quotes
- [**Ireg**] represents data found at address in Ireg
- **fmt** represents a format
  - input: INT, FLT, CHR, STR
  - output: INT, FLT, CHR, STR, HEX, PTR

kcode instruction set

LDA Ireg =lbl	Ireg $\leftarrow$ address(lbl)
LDR reg Ireg	reg $\leftarrow$ [Ireg]
STR reg Ireg	reg $\rightarrow$ [Ireg]
MOV reg_a reg_b	$\text{reg}_a \leftarrow \text{reg}_b$
PUSH reg	top stack $\leftarrow$ reg
POP reg	reg $\leftarrow$ top stack
I2F Freg Ireg	Freg $\leftarrow$ Ireg
F2I Ireg Freg	Ireg $\leftarrow$ Freg
ADD Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b + \text{Ireg}_c$
SUB Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b$ - Ireg_c
MUL Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b$ * Ireg_c
DIV Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b / \text{Ireg}_c$
MOD Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \% \text{Ireg}_c$
NEG Ireg	Ireg $\leftarrow$ - Ireg
UADD Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b + \text{Ireg}_c$
USUB Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b$ - Ireg_c
UMUL Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b$ * Ireg_c
UDIV Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b / \text{Ireg}_c$
UMOD Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \% \text{Ireg}_c$
FADD Freg_a Freg_b Freg_c	$\text{Freg}_a \leftarrow \text{Freg}_b + \text{Freg}_c$
FSUB Freg_a Freg_b Freg_c	$\text{Freg}_a \leftarrow \text{Freg}_b$ - Freg_c
FMUL Freg_a Freg_b Freg_c	$\text{Freg}_a \leftarrow \text{Freg}_b$ * Freg_c
FDIV Freg_a Freg_b Freg_c	$\text{Freg}_a \leftarrow \text{Freg}_b / \text{Freg}_c$
FNEG Freg	Freg $\leftarrow$ - Freg
BAND Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \,\&\, \text{Ireg}_c$
BOR Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \,\|\, \text{Ireg}_c$
BNOT Ireg	Ireg $\leftarrow \sim$ Ireg
BXOR Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \wedge \text{Ireg}_c$
LAND Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \,\&\&\, \text{Ireg}_c$
LOR Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \mid \text{Ireg}_c$
LNOT Ireg	Ireg$\leftarrow$! Ireg
LXOR Ireg_a Ireg_b Ireg_c	$\text{Ireg}_a \leftarrow \text{Ireg}_b \otimes \text{Ireg}_c$

JMP lbl	PC $\leftarrow$ lbl
CMP Ireg$_a$ Ireg$_b$	signed integer comparison
UCMP Ireg$_a$ Ireg$_b$	unsigned integer comparison
FCMP Freg$_a$ Freg$_b$	floating point comparison
JEQ lbl	PC $\leftarrow$ lbl (if EQ flag set)
JNE lbl	PC $\leftarrow$ lbl (if EQ flag not set)
JGT lbl	PC $\leftarrow$ lbl (if GT flag set)
JGE lbl	PC $\leftarrow$ lbl (if LT flag not set)
JLT lbl	PC $\leftarrow$ lbl (if LT flag set)
JLE lbl	PC $\leftarrow$ lbl (if GT flag not set)
ROL Ireg imm$_6$	rotate Ireg left
ROR Ireg imm$_6$	rotate Ireg right
SHL Ireg imm$_6$	shift Ireg left
SHR Ireg imm$_6$	shift Ireg right
ASHL Ireg imm$_6$	arithmetic shift Ireg left
ASHR Ireg imm$_6$	arithmetic shift Ireg right
CALL lbl	RP $\leftarrow$ PC; PC $\leftarrow$ lbl
RET	PC $\leftarrow$ RP
NOP	do nothing!
HALT	stop!
INC Ireg	Ireg $\leftarrow$ Ireg + 1
DEC Ireg	Ireg $\leftarrow$ Ireg - 1
GET fmt	top stack $\leftarrow$ read value
GETLN	read upto/including carriage return
PUT fmt	top stack $\rightarrow$ write value
PUTLN	print carriage return
MALLOC Ireg$_a$ Ireg$_b$	Ireg$_b$ is size; Ireg$_a$ $\leftarrow$ address
DALLOC Ireg$_a$ Ireg$_b$	Ireg$_b$ is size; Ireg$_a$ is address

1.4 Why VISUAL_KBOX?

I think a very obvious question at this point would be: ”Why are we talking about something previously done?”

And my answer is: ”Because!”

My first reason to say ”because” is that I was not completely happy with my first attempt at implementing a hypothetic assembly language that could possibly be used as an education piece of software. The main flaw that existed was that the user of the software was only experiencing the typical cycle: write code, assemble code, run code with the fourth ever-present element – why didn’t it run correctly! The **kbox** computer with its **kcode** assembly language very much served its purpose to help students understand programming languages and compiler construction. But I wished it had also clarified how some of the low-level aspects of computing actually worked. I wanted the programmer not only to see the result but also to see incremental steps in getting to that result. I wanted them to see the registers, see the instruction to be executed, to see the resulting change.

My second reason to say ”because” is that I already had written most of the code and it was working pretty well. All it really required was the implementation of a graphical user interface. But that requirement was a VERY BIG BUT for me. I have never programmed in any sort of graphical programming language. I would have to learn everything from scratch.

And that became my third reason to say ”because”. My major obstacle became my primary motivation in the new project.

I decided I wanted to implement a graphical user interface for KBOX which would highlight the inner workings of a machine while I simultaneously cleaned up some of my previous coding and hopefully learned some new programming skills.

The remainder of this text is my experience in completing the project. The organization of material essentially parallels my approach to upgrading my previous implementation:

- The First Phase is to rewrite my previous code. I chose to switch from writing C source code back to my original programming language C++. I chose to do so to have the advantage of object-oriented features and many of the graphical tool kits seemed to favor C++.
- The Second Phase is to implement the graphical user interface. I ultimately chose to work with FLTK, the Fast Light Toolkit. This came only after a lot of blood, sweat, and tears installing and testing several different such toolkits. When you do not know what you are doing, you do a lot of trial and error!
- The Third Phase is to setup the **kbox** simulator to execute the desired source code file. This actually requires reading in the entire program from the source file, recognizing and processing the data segment and the code segment, using the data segment to define a memory map for static/global variables during the simulation, using the code segment to build the list of instructions and maintain the list of important label locations, and lastly initiating all registers and flags in preparation for actually executing the code
- The Fourth Phase is to define the **fetch-decode-execute cycle** that will actually execute the individual instructions in the proper order. This phase, as the previous phase, must interface with both the underlying **kbox** implementation code and the **kwindows** graphical user interface code to execute instructions correctly and to display results properly.
- The Last Phase is almost anti-climactic. A main driver is needed to put all this machinery in motion.

Chapter 2

KBOX Implementation

The KBOX Vaping Machine!

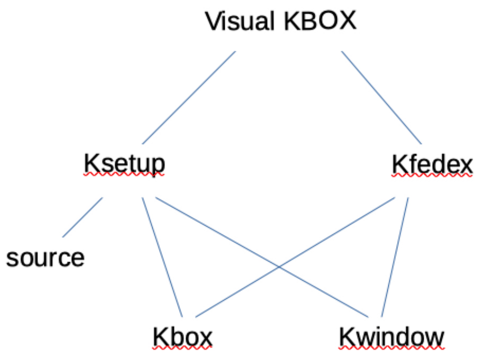
2.1 Overview

The original implementation of the **kcode** simulator was a number of C source code files with a main driver to supervise proper setup and execution. This new implementation will essentially be the same!

However, I have moved away from C coding to C++ coding. I personally have a slight preference for C++ over C. Plus, as I mentioned previously graphical toolkits seemed to have a similar preference.

Another obvious modification I chose to make was to combine all the C implementation files specific to the **kbox** architecture into a single C++ implementation (i.e., a single header file together with a single implementation file). Previously each component in the **kbox** was grouped together in its own unique implementation. Now each component is defined as its own class with these classes being combined in a collective implementation. A single header file declares all classes, their members and their methods; a single implementation defines the method algorithms.

We will summarize the **kbox** source code shortly. But before that, the following diagram illustrates the **visual_kbox** coding hierarchy.



2.2 Preliminary Comments

My original implementation of **kbox** was focused on a large number of implementation files:

- ktypes
- kflags
- memory
- registers
- kcode
- ksetup
- ksource
- kbox

The last file was the main driver which coordinated all communication between all the others.

In my current implementation of **visual_kbox** I have tried to reduce the number of source files by combining the first five files into a single pair (header and implementation file). I have also attempted to reduce the number of methods to the minimum. Using C++ classes and defining methods within these classes allowed for consistent names for similar functionality and shorter argument lists. This hopefully will make it easier to remember although it also requires typing longer object qualified identifiers for referencing class methods.

All C source code for my original implementation is available for download at:

www.cs.lewisu.edu/∼kaiserpa/books/kbox/resources

All C++ source for my new implementation is also available for download at:

www.cs.lewisu.edu/∼kaiserpa/books/vkbox/resources

Chapter 3

KBOX Source Code

KBOX Karaoke!

3.1 Overview

As I mentioned in the very last chapter, I have attempted to combine five of my original C implementations for **kbox** into a single implementation defining all the C++ classes and structs that will be used throughout the simulator.

This section summarizes much of the detail found in this consolidation. The original algorithms have pretty much been carried over intact.

Five previous header files are now combined into the single C++ header file **kbox.h**. This header file declares the following items:

global constants

Three global constants are declared and defined:

- literal constants are represented as strings
  const long long int LITERAL_SIZE = 256;
- memory storage capacity
  const long long int MEMORY_SIZE = 1000000;
- register capacity
  const long long int NUMBER_REGISTERS = 16;

kbox types

Five basic types are available in **kbox**: klunk, int64, flt64, chr64, and str64.

- int64: binary representation of a signed integer
  using twos-complement
- flt64: binary representation of a signed floating point number using IEEE 64-bit binary representation
- chr64: ASCII code representation of single character
  ASCII pattern is in first 8 bits; remaining 56 bits are undefined
- str64: sequence of characters terminated by '\0'
  in reality it is a C++ pointer to a dynamic string data value!
- klunk: binary representation of an unsigned integer
  represents raw bit patterns, non-negative integers, and **kbox** pointers

All movement of data within the **kbox** simulator is done with klunks! Only when data is actually involved in a calculation or a comparison is the klunk converted to its actual data type and manipulated.

The methods declared in the header file provide conversion algorithms to easily move between the five basic types above. In C++ it is a reinterpret-cast on the address of the klunk.

In addition, two other conversion algorithms are provided:

- literal2klunk: which converts an immediate value in a **kcode** instruction into a C++ string
- klunk2hex: which converts a bit pattern into 16-character hexadecimal string for display

kbox memory

Memory in the **kbox** simulator is a simple array of klunks! MEMORY_SIZE has been set at 1000000. So far this has been sufficient for the programs that I have tested using the **kcode** assembly language directly and also using the **kize** compiler (**Fun With Programming Languages**). But at any time in the future it may be found to be wanting!

Unfortunately, a simple array of klunks is not the whole story! Keeping track of global variables and static storage requires assigning and remembering specific locations in memory for the storage and retrieval. Hence, a memory map (table or list) must be maintained, including:

- identifer: the name of the global / static variable
- location: its address in the memory array
- size: the number of klunks required for storage

The methods declared in the header file provide provide access to the memory map inserting and retrieving entries and also access to memory array itself.

I specifically included two commands in **kcode** for inserting and retrieving data into and from memory:

- void poke (klunk address,klunk data);
- klunk peek (klunk address) const;

as a homage to the most popular original personal computer – **the Commodore 64**.

kbox registers

The registers (or accumulators) are where all the real action takes place. All calculations and comparisons are done in the registers. Registers come in two flavors: integer (which really means bit patterns) and floating point (which really means decimals).

For **kbox** the register sets are limited to NUMBER_REGISTERS = 16, which is probably a bit small, especially for a totally hypothetical architecture! However, sixteen seemed completely satisfactory for covering basic programming and compiler construction techniques. There is no reason why this constant can not be increased; I just felt that sixteen was sufficient for my needs.

kbox flags

The **kbox** flag set is most minimal! The program counter (PC) keeps track of the location of the next instruction to be executed by the computer. Only three other flags are implemented to show the outcome of the most recent comparison instruction:

- EQ: both values are equal
- GT: indicates the first value is larger than the second
- LT: indicates the second value is larger than the first

Only one of these three flags will be set to **true** at any given time; the remaining two flags would be set to **false**.

Most assemblers provide a wide range of additional flags to identify a variety of circumstances that might occur during execution. **Kbox** is not intended to be comprehensive in scope but helpful in learning the fundamentals.

The methods declared in the header file are very elementary and require no explanation.

kbox code

The executable code for a **kcode** program is simply a list of instructions. The position within the list is important for branching statements so that transfer of control can be properly implemented. Simple branching statements (either an unconditional jump or a conditional jump) move to a different location within the code with no intention of coming back; the **call** and **ret** statements, however, provide a basic mechanism for a program to move to a different location but come back and continue where it left off.

The actual instructions are basically four-tuples: an opcode followed by up to three operands.

All this is quite straight-forward. The minor complication is that other lists are also associated with this basic instruction list:

- global declarations: what identifiers found within this file need to be shared with the outside world
- extern declarations: what identifiers found in the outside world need to be known within this file
- label declarations: what internal identifiers will be used by branching instructions

The first two items are really unnecessary in the **kbox** simulation. All **kbox** programs must be self-contained. The can not be linked to other code files; the only global declaration often required in assembly language is the location of the first instruction (often called **main**). By default **kbox** begins execution with the first instruction. The use of the global declaration is highly encouraged but in reality it does nothing.

Lastly, **kbox** label declarations require the use of the _LABEL directive, which is not found in other assembly languages. I chose to incorporate it in the **kcode** assembly language for two reasons:

- to highlight identifying the key locations within the instruction code
- to simplify the parsing of the assembly language – all directives start with the underscore (_) character

kbox struct

The final element in the header file is the declaration of a simple struct (all its members public), one member for each of the component classes above.

- MEMORY
- REGISTERS
- FLAGS
- INSTRUCTIONS
- GLOBALS
- EXTERNS
- LABELS

3.2 kbox.h

kbox.h source code

```
#ifndef _KBOX_H
#define _KBOX_H

#include <cstdlib>
#include <string>
#include <map>
#include <vector>

using namespace std;

/* ---------------------------------------------------------------- */

const long long int LITERAL_SIZE        = 256;
const long long int MEMORY_SIZE         = 1000000;
const long long int NUMBER_REGISTERS    = 16;

/* ---------------------------------------------------------------- */
/*
*/
/*                              ktypes
*/
/*
*/
/* ---------------------------------------------------------------- */

typedef unsigned long long int  klunk;

typedef long long int           int64;
typedef double                  flt64;
typedef char                    chr64;
typedef char*                   str64;

int64 klunk2int (klunk);        // klunk -> int64
flt64 klunk2flt (klunk);        // klunk -> flt64
chr64 klunk2chr (klunk);        // klunk -> chr64
str64 klunk2str (klunk);        // klunk -> str64

klunk int2klunk (int64);        // int64 -> klunk
klunk flt2klunk (flt64);        // flt64 -> klunk
klunk chr2klunk (chr64);        // chr64 -> klunk
klunk str2klunk (str64);        // str64 -> klunk

klunk literal2klunk (const string&);

char* klunk2hex (klunk);

/* ---------------------------------------------------------------- */
```

```
/*
*/
/*                              kflags
*/
/*
*/
/* ———————————————————————————————————————————————————————— */

class flags_type
{
private:
  klunk          PC;
  bool           EQ;
  bool           GT;
  bool           LT;

public:
  flags_type(klunk pc=1,bool eq=true,
    bool gt=false,bool lt=false)
  : PC(pc),EQ(eq),GT(gt),LT(lt)
  { }

  klunk getPC (void) const
  { return PC; }
  bool getEQ (void) const
  { return EQ; }
  bool getGT (void) const
  { return GT; }
  bool getLT (void) const
  { return LT; }

  void setPC (klunk k)
  { PC = k; }
  void setEQ (void)
  { EQ = true; LT = GT = false; }
  void setGT (void)
  { GT = true; EQ = LT = false; }
  void setLT (void)
  { LT = true; EQ = GT = false; }

  void display (void) const;
};

/* ———————————————————————————————————————————————————————— */
/*
*/
/*                              kmemory
*/
/*
*/
/* ———————————————————————————————————————————————————————— */

class map_entry
```

```
{
private:
  string        name;
  klunk         address;
  klunk         size;

public:
  map_entry ()
  : name(""),address(0),size(0)
  {  }
  map_entry (const string& n,klunk a,klunk s)
  : name(n),address(a),size(s)
  {  }

  string get_name (void) const
  { return name; }
  klunk get_address (void) const
  { return address; }
  klunk get_size (void) const
  { return size; }

  void set_name (const string& n)
  { name = n; }
  void set_address (klunk a)
  { address = a; }
  void set_size (klunk s)
  { size = s; }
};

class memory_type
{
private:
  klunk         data[MEMORY_SIZE];
  klunk         next_available;

public:
  vector<map_entry>     map;

  memory_type (void)
  : next_available(1)
  {  }

  klunk get_next_available (void) const
  { return next_available; }
  void set_next_available (klunk k)
  { next_available = k; }

  klunk peek (klunk k) const
  { return data[k]; }
  void poke (klunk k,klunk val)
  { data[k] = val; }

  bool is_map_entry (const string&,map_entry&) const;
  void insert_map_entry (const string&,klunk,klunk);

  void define (const string&,klunk);
  void reserve (const string&,klunk);

  void display_map (void) const;
  void display_static (void) const;
  void display_stack (klunk k) const;
  void display_heap (klunk k) const;
};

/* ————————————————————————————————————————— */
/*
*/
/*                    kregisters
*/
/*
*/
/* ————————————————————————————————————————— */

class registers_type
{
private:
  klunk          Iregisters[NUMBER_REGISTERS];
  klunk          Fregisters[NUMBER_REGISTERS];

public:
  bool is_register (string&,char&,klunk&) const;

  klunk get_register (string&) const;
  void set_register (string&,klunk);

  void display_Iregisters (void) const;
  void display_Fregisters (void) const;
};

/* ————————————————————————————————————————— */
/*
*/
/*                    kcode
*/
/*
*/
/* ————————————————————————————————————————— */

class global_table
{
private:
  vector<string>         data;

public:
  bool is_global (const string&) const;
  void add_global (const string& n)
  { data.push_back(n); }
  void display (void) const;
};

class extern_table
{
private:
  vector<string>        data;

public:
  bool is_extern (const string&) const;
  void add_extern (const string& n)
  { data.push_back(n); }
  void display (void) const;
};

class label_entry
{
private:
  string          name;
  klunk           location;

public:
  label_entry (const string& n,klunk l)
  : name(n),location(l)
  {  }

  string get_name (void) const
  { return name; }
  klunk get_location (void) const
  { return location; }
};

class label_table
{
private:
  vector<label_entry>   data;

public:
  bool is_label (const string&) const;
  void add_label (const label_entry& le)
  { data.push_back(le); }
  void display (void) const;
  klunk get_location (const string&) const;
};

class code_entry
{
private:
  string          opcode;
  string          operand1;
  string          operand2;
  string          operand3;
```

```
public:
  code_entry (const string& opc,const string& op1,
              const string& op2,const string& op3)
  : opcode(opc),operand1(op1),operand2(op2),operand3(op3)
  {  }

  string get_opcode (void) const
  { return opcode; }
  string get_operand1 (void) const
  { return operand1; }
  string get_operand2 (void) const
  { return operand2; }
  string get_operand3 (void) const
  { return operand3; }
};

class code_table
{
private:
  vector<code_entry>    data;

public:
  klunk get_code_size (void) const
  { return data.size(); }

  code_entry get_instruction (klunk k) const
  { return data[k]; }

  void add_instruction (const code_entry& ce)
  { data.push_back(ce); }
};

/* ---------------------------------------------------------- */
/*
*/
/*                          KBOX
*/
/*
*/
/* ---------------------------------------------------------- */

struct kbox_type
{
  memory_type           MEMORY;
  registers_type        REGISTERS;
  flags_type            FLAGS;
  code_table            INSTRUCTIONS;
  global_table          GLOBALS;
  extern_table          EXTERNS;
  label_table           LABELS;
};
```

VISUAL KBOX

```
#endif // _KBOX_H
```

3.3 kbox.cpp

kbox.cpp source code

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <string>
#include <cstring>

#include "kbox.h"

using namespace std;

/* ------------------------------------------------------------------ */

int64 klunk2int (klunk k)                   // klunk -> int64
{  return *(reinterpret_cast<int64*>(&k));  }

flt64 klunk2flt (klunk k)                   // klunk -> flt64
{  return *(reinterpret_cast<flt64*>(&k));  }

chr64 klunk2chr (klunk k)                   // klunk -> chr64
{  return *(reinterpret_cast<chr64*>(&k));  }

str64 klunk2str (klunk k)                   // klunk -> str64
{  return (reinterpret_cast<str64>(k));  }

klunk int2klunk (int64 i)                   // int64 -> klunk
{  return *(reinterpret_cast<klunk*>(&i));  }

klunk flt2klunk (flt64 f)                   // flt64 -> klunk
{  return *(reinterpret_cast<klunk*>(&f));  }

klunk chr2klunk (chr64 c)                   // chr64 -> klunk
{  return *(reinterpret_cast<klunk*>(&c));  }

klunk str2klunk (str64 s)                   // str64 -> klunk
{  return (reinterpret_cast<klunk>(s));  }

klunk literal2klunk (const string& lit) // literal -> data
{
  char symbol = lit[0];
  if (symbol == '\"')
  // string literal
  {
    // remove leading and trailing '\"'
    str64 result = (char*) malloc (LITERAL_SIZE);
    int new_len = lit.size()-2;
    for (int i = 0;i < new_len;i++)
      result[i] = lit[i+1];
    result[new_len] = '\0';
    return str2klunk (result);
  }
  else if (symbol == '\'')
  // character literal
    return chr2klunk (lit[1]);
  else if ((symbol == '+') ||
           (symbol == '-') ||
           (isdigit(symbol)))
  // numeric literal
  if (lit.find_first_of(".eE") != string::npos)
    // floating point literal
      return flt2klunk (stof(lit));
    else
    // integer literal
      return int2klunk (stoll(lit));
  else
  {
    cerr << "invalid literal encountered: "
         << lit << endl;
    exit (EXIT_FAILURE);
  }
}

char* klunk2hex (klunk k)                 // klunk -> hex
{
  static const char* HEX_DIGITS = "0123456789ABCDEF";
  char* result = new char[17];
  result[16] = '\0';
  for (int i=0;i<16;i++)
  {
    result[15-i] = HEX_DIGITS[k%16];
    k = k/16;
  }
  return result;
}

/* ---------------------------------------------------------- */

void flags_type::display (void) const
{
  cout << setw(5) << left << "PC"
       << setw(5) << left << getPC()
       << setw(5) << left << "EQ"
       << setw(5) << left << getEQ()
       << setw(5) << left << "GT"
       << setw(5) << left << getGT()
       << setw(5) << left << "LT"
       << setw(5) << left << getLT()
       << endl << endl;
}

/* ---------------------------------------------------------- */
```

```
bool memory_type::is_map_entry
  (const string& n,map_entry& me) const
{
  for (klunk i=0;i<map.size();i++)
    if (map[i].get_name() == n)
    {
      me = map[i];
      return true;
    }
  return false;
}

void memory_type::insert_map_entry
  (const string& n,klunk a,klunk s)
{
  map_entry me(n,a,s);
  map.push_back(me);
}

void memory_type::define
  (const string& name,klunk val)
{
  map_entry me;
  if (is_map_entry(name,me))
  {
    cerr << "duplicate identifier ("
         << name << ") encountered!\n";
    return;
  }
  klunk address = get_next_available();
  insert_map_entry(name,address,1);
  poke(address,val);
  set_next_available(get_next_available()+1);
}

void memory_type::reserve
  (const string& name,klunk size)
{
  map_entry me;
  if (is_map_entry(name,me))
  {
    cerr << "duplicate identifier ("
         << name << ") encountered!\n";
    return;
  }
  klunk address = get_next_available();
  insert_map_entry(name,address,size);
  set_next_available(get_next_available()+size);
}

void memory_type::display_map (void) const
{
  cout << "GLOBAL/STATIC MEMORY MAP:" << endl;
  for (klunk i=0;i<map.size();i++)
    cout << setw(20) << left  << map[i].get_name()
         << setw(10) << right << map[i].get_address()
         << setw(10) << right << map[i].get_size()
         << endl;
  cout << endl;
};
void memory_type::display_static (void) const
{
  cout << "GLOBAL/STATIC MEMORY:" << endl;
  for (klunk i=0;i<get_next_available();i++)
    cout << setw(10) << right << i
         << setw(20) <<right << klunk2hex(peek(i))
         << endl;
  cout << endl;
};

void memory_type::display_stack (klunk sp) const
{
  cout << "STACK:" << endl;
  for (klunk i=MEMORY_SIZE-1;i>=sp;i--)
    cout << i << "    " << klunk2hex(peek(i)) << endl;
  cout << endl;
}

void memory_type::display_heap (klunk hp) const
{
  cout << "HEAP:" << endl;
  for (klunk i=get_next_available();i<hp;i++)
    cout << i << "   " << klunk2hex(peek(i)) << endl;
  cout << endl;
}

/* ----------------------------------------------------------- */

bool registers_type::is_register
  (string& name,char& kind,klunk& index) const
{
static map<string,string> ALIASES =
  {
    {"IA","I0"},{"IB","I1"},{"IC","I2"},{"ID","I3"},
    {"SAR","I4"},{"DAR","I5"},{"OR","I6"},{"IR","I7"},
    {"ZR","I11"},
    {"SP","I12"},{"FP","I13"},{"RP","I14"},{"HP","I15"},
    {"FA","F0"},{"FB","F1"},{"FC","F2"},{"FD","F3"}
  };

  for (auto iter=ALIASES.cbegin();
       iter!=ALIASES.cend();
       iter++)
    if (iter->first == name)
      name = iter->second;
  kind = name[0];
  if ((kind != 'I') && (kind != 'F'))
  {
    kind = 'X';
    index = 0;
    return false;
  }
  try
  {
    index = stoull(name.substr(1,name.size()-1));
    if (0 <= index < 16)
      return true;
    else
    {
      kind = 'X';
      index = 0;
      return false;
    }
  }
  catch (const runtime_error& e)
  {
    kind = 'X';
    index = 0;
    return false;
  }
}

klunk registers_type::get_register
  (string& name) const
{
  char kind;
  klunk index;
  if (is_register(name,kind,index))
    if (kind == 'I')
      return Iregisters[index];
    else
      return Fregisters[index];
  else
  {
    cerr << "invalid register (" << name << ")\n";
    exit(EXIT_FAILURE);
  }

}
void registers_type::set_register
  (string& name,klunk val)
{
  char kind;
  klunk index;
  if (is_register(name,kind,index))
    if (kind == 'I')
      Iregisters[index] = val;
    else
      Fregisters[index] = val;
```

```
}

void registers_type::display_Iregisters (void) const
{
  cout << "I registers" << endl;
  for (int i=0;i<NUMBER_REGISTERS; i++)
    cout << setw(5) << left << "I"+to_string(i)
         << setw(16) << left << klunk2hex(Iregisters[i])
         << endl;
  cout << endl;
}

void registers_type::display_Fregisters (void) const
{
  cout << "F registers" << endl;
  for (int i=0;i<NUMBER_REGISTERS; i++)
    cout << setw(5) << left << "F"+to_string(i)
         << setw(16) << left << klunk2hex(Fregisters[i])
         << endl;
  cout << endl;
}

/* ------------------------------------------------------------ */

bool global_table::is_global (const string& n) const
{
  for (klunk i=0;i<data.size(); i++)
    if (data[i] == n)
      return true;
  return false;
}

void global_table::display (void) const
{
  cout << "GLOBAL TABLE:" << endl;
  for (klunk i=0;i<data.size(); i++)
    cout << setw(20) << left << data[i]
         << endl;
  cout << endl;
}

bool extern_table::is_extern (const string& n) const
{
  for (klunk i=0;i<data.size(); i++)
    if (data[i] == n)
      return true;
  return false;
}

void extern_table::display (void) const
{
  cout << "EXTERN TABLE:" << endl;
  for (klunk i=0;i<data.size(); i++)
```

```
    cout << setw(20) << left << data[i]
         << endl;
  cout << endl;
}

bool label_table::is_label (const string& n) const
{
  for (klunk i=0;i<data.size();i++)
    if (data[i].get_name() == n)
      return true;
  return false;
}

void label_table::display (void) const
{
  cout << "LABEL TABLE:" << endl;
  for (klunk i=0;i<data.size();i++)
    cout << setw(20) << left << data[i].get_name()
         << setw(10) << left << data[i].get_location()
         << endl;
  cout << endl;
}

klunk label_table::get_location (const string& n) const
{
  for (klunk i=0;i<data.size();i++)
    if (data[i].get_name() == n)
      return data[i].get_location();
  return 0ULL;
}
```

Chapter 4

KWINDOWS Implementation

The KBOX Home!

4.1 Overview

This chapter is really the reason for my writing this text. I was very happy with my original **kcode** simulator and how it worked with my compiler construction material.

But I thought **kbox** and **kcode** would be an even better educational tool if one could actually see the result of stepping through an assembly language program. That would require a graphical user interface to be incorporated into my previous coding. And that is the focus of this chapter.

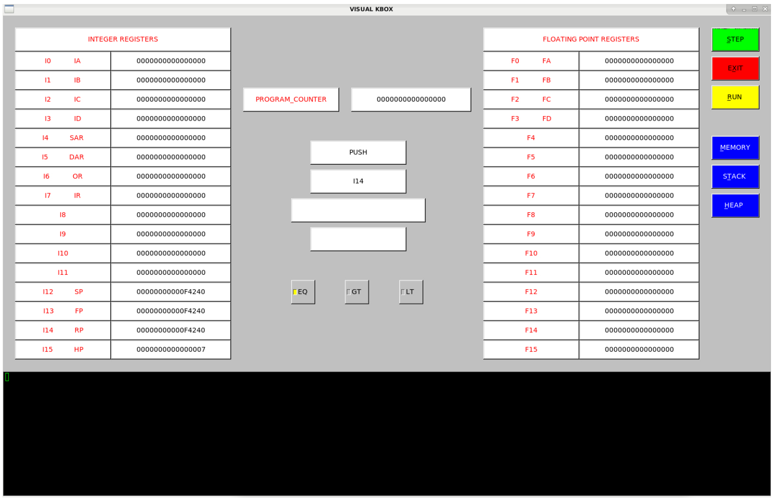
The following page contains an image of how I would want such an interface to look:

- integer registers are displayed in a column on the left-hand side
- floating point registers are displayed in a column on the right-hand side
- the program counter appears in the top center of the screen
- the opcode and operands appear beneath
- the three comparison flags are found at the bottom center
- six command buttons are displayed on the extreme right-hand side
- the very bottom is a console display of program progress

The command buttons are organized into two sets of three:

- step button: execute the current instruction
- exit button: terminate execution
- run button: execute the instruction set to completion

- memory button: display current static/global memory
- stack button: display current stack contents
- heap button: display current heap contents



VISUAL KBOX

INTEGER REGISTERS	
I0 IA	0000000000000000
I1 IB	0000000000000000
I2 IC	0000000000000000
I3 ID	0000000000000000
I4 SAR	0000000000000000
I5 DAR	0000000000000000
I6 OR	0000000000000000
I7 IR	0000000000000000
I8	0000000000000000
I9	0000000000000000
I10	0000000000000000
I11	0000000000000000
I12 SP	00000000000F4240
I13 FP	00000000000F4240
I14 RP	00000000000F4240
I15 HP	0000000000000007

PROGRAM_COUNTER	0000000000000000

FLOATING POINT REGISTERS	
F0 FA	0000000000000000
F1 FB	0000000000000000
F2 FC	0000000000000000
F3 FD	0000000000000000
F4	0000000000000000
F5	0000000000000000
F6	0000000000000000
F7	0000000000000000
F8	0000000000000000
F9	0000000000000000
F10	0000000000000000
F11	0000000000000000
F12	0000000000000000
F13	0000000000000000
F14	0000000000000000
F15	0000000000000000

I found this particular project a bit more difficult than my previous programming projects. Difficult for several different reasons:

First, I have never previously done any sort of graphical programming in any programming language. I assumed there would be some subtle differences that would arise – and I was not disappointed!

Second, selecting the graphics toolkit to work with was more difficult than I had anticipated. When you know nothing and want to start, you have to figure out what is good software – good for the project and good for your ability. I first tried Qt, then I considered GTK, then I considered FLTK, then I quit. All the tutorials I was reading did a "Hello, World!" program and then seemed to move directly into writing a word processor with drop-down menus, popup menus, and little explanation of what was really taking place.

Third, most resources appeared to me to be more reference manual rather than introductory tutorial. I certainly was floundering and was uncertain what to do.

I then remembered a very good textbook I had encountered several years back which had several chapters on graphical user interfaces – Bjarne Stroustrup, **Programming Principles and Practice Using C++, 2nd edition**, Addison-Wesley, 2014, ISBN 978-0-321-99278-9.

I really liked his presentation and explanations. His use of FLTK, the **fast light toolkit**, was my motivation to focus of FLTK for my project. And even though his version of FlTK was not the same as my version and had several significant differences, I was resolved to stay with this toolkit throughout the duration of this project.

Much of the exposition which follows here summarizes key points in Stroustrup's book. Please refer to **Programming Principles and Practice Using C++** for a more detailed (and probably better written) explanation. But the following elements proved very helpful to me in transitioning to programming with graphical user interfaces.

categories of interfaces

User interfaces for input and output typically fall into one of the following three categories:

- console: utilization of the command line and emphasis on textual data
- graphical: visual screen display together with mouse, touch-screen, keyboard
- web browser: dynamic pages utilizing a combination of a markup language (HTML) and a scripting language (Perl)

Perhaps the biggest difference between the traditional console interface and the others is the significant reversal of primary roles between the application and the user of the application.

With a console interface, the typical input sequence has the following form:

application ⟶ input prompt ⟶ user response

However, with a graphical interface, the typical input sequence has the following form:

user action ⟶ system callback ⟶ application response

This reversal of roles and unfamiliarity with its requirements makes program organization, development, and debugging more difficult than the familiar console interface.

Two very basic admonitions immediately come to mind:

- **KISS** – Keep It Simple, Stupid!
  keep the graphical user interface definition simple and basic
- work incrementally
  build the graphical user interface incrementally and
  test thoroughly before moving to the next step

4.2 Windows, Widgets, and Whatever

In this section I present a brief overview of terminology and organization of graphical components as I currently understand them. Like many readers, I am a newbie to this topic. I now know enough to be dangerous, but I also do not know enough to say some things with certainty. Feel free to send me any corrections or better explanations for any mis-steps which may follow!

First of all, a graphical display area is referred to as a **window**. It is a rectangular region in which to display other elements.

These other elements can include a variety of options:

- boxes, either text or numeric
- buttons
- selections, exactly one from many or several from many
- menus, either drop-down or pop-up
- other: sliders, dials, etc.

All these elements are general lumped into one very large collective category called **widget**.

Like so many other concepts in computer science, a widget may also have a recursive property in its elements. That is, a widget might also include a an element which is itself a collection of other widgets. This parallels other very common structures, such as folders containing subfolders, algorithms containing subalgorithms, or a simple list being either empty or a single item followed by another simple list.

I will say right off that I have not yet worked with aggregate widgets. My graphical user interface will be built essentially with a single window and several widgets as immediate children of that window. I am comfortable that this approach is sufficient for the project at hand. But as a teacher this topic must be placed somewhere on my **bucket list** of things to do!

I highly recommend having graph paper (or simple scrap paper) handy with a ruler, pencil, and eraser for developing your graphical user interface. In addition, I would not attempt to implement a full blown graphical interface in one attempt! I recommend as Bjarne Stroustrup does, develop the interface in small incremental steps.

My brief experience has been that the best diagrams on paper when brought to the computer screen look very different (and not for the better). I found a best guess, followed by displaying the result, followed by an adjustment, followed by displaying the result, ... seemed faster in the long run.

Lastly, just about every graphical toolkit will have its own version of a graphical user interface developer. In FLTK this developer is called FLUID.

I have not really investigated FLUID in any great detail. This is another topic that has been placed on my **bucket list** of things to do. I believe it will be a very useful tool and a very significant time saver with future graphical projects. But at this point in my journey I am more concerned about understanding the process and the building blocks than accelerating the speed of the process. So all my FLTK source code will be of my own design and not computer generated.

widget requirements

Although widgets come in a variety of flavors and options, widgets share five common characteristics:

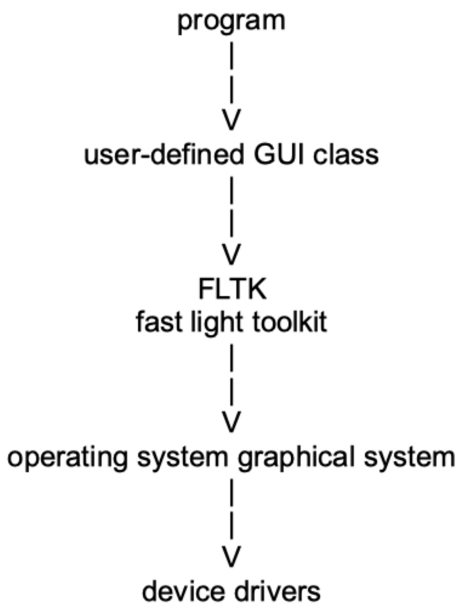
- the widget must be defined
    the type of widget and its properties
- the widget must be displayed
    it may be visible or it may be hidden
    if modifications are made, it must be displayed again
- define what to do if the widget is ”activated”
    activation may take many forms:
        e.g., mouse click or mouse hover
    utilizes a ”call-back” mechanism
- define what to do when notified
    what should the program do
        when it receives a ”call-back”
- the widget then basically waits for a call-back

The third and fourth characteristics look a bit strange in C++ coding. The two are closely related. First, the system must recognize that some activity has taken place and notify the application something just happened; but this must be done at a very low level. Even though our source code is in C++ (or another high level language) may have all sorts of bells and whistles, the system response is very vanilla; and our ”call-back” mechanism must be very basic as well. Essentially the sole purpose of the ”call-back” mechanism is to capture the system notification and then immediately convert it to an actual call statement to the real event handler.

The real even handler will be part of our high level source and may make use of the bells and whistles the particular programming language provides.

Hopefully this will become clear when we actually consider C++ source code for our graphical user interface. Stroustrup’s book provides a very good template for implementing the five characteristics for a widget. I incorporated this template into the source code which follows shortly.

The following diagram illustrates the various levels that comprise a graphical user interface: from the program, to a user-defined graphics class, to FLTK, to the operating system graphical system, to the actual device drivers.



The call-back mechanism starts at the lowest level and propagates upward to the call-back function defined in the GUI class by the programmer.

4.3 KBOX Display (Part I)

I first want to summarize the components I chose to incorporate into the **visual_kbox** display.

- integer registers:
  - label on the top
  - descriptive label for each
  - hexadecimal value for each
- floating point registers:
  - similar display as integer registers
- program counter, current instruction, and flags
  - between the two register sets
- six command buttons
  - along right-hand edge
  - step, run, and exit
    - change window contents as appropriate
  - memory, stack, and heap
    - show in console display at the bottom

The following summary, for better or worse, describes my incremental implementation of this design. I say for better or worse because I too have been learning graphical user interfaces from scratch. Much of what I did was trial and error, and frustration, a lot of frustration! But it does eventually come together. Hopefully some of you readers will send me additional insights on improving my coding and/or explaining things better.

- plan the layout
- display the command buttons on right-hand edge
- display the integer registers on the left-hand edge
- display the floating point registers to the right
- display the program counter and the instruction
- display the flags
- display the console at the bottom of the screen
- get the call-back mechanism working
  for step, exit, and lastly run
- get the call-back mechanism working
  for memory, stack, and heap

Let us now examine each of the above items in more detail.

layout

Graph paper or scrap paper is especially useful. The important thing is to visualize what you want and put the design on paper.

All widgets in FLTK are basically built around four integer values **(x , y , w , h)**.

x represents the x-axis offset from the origin (0,0), i.e., horizontal
y represents the y-axis offset from the origin (0,0), i.e., vertical
w represents the width of the widget (horizontal)
h represents the height of the widget (vertical)

The origin (0,0) represents the **top-left** corner of the screen. Everything in FLTK is essentially built moving **down** and **to the right**. Down and to the right . . . ; down and to the right . . .; down and to the right . . .

I feel like I am trapped in an Oliver Stone movie.

The C++ declarations for FLTK widgets will generally require at least these four integers. In addition, specific widget categories will each have their own members and methods which must be learned. However, we can not learn everything in one sitting; we focus on a few basic widgets and expand that base gradually.

The first objective should be getting an attractive display up and running on the screen, even if filled with artificial dummy data. Once we have a basic display, we can tweak it and gradually add more functionality.

One of the first items you will notice in the **kwindow** header file is the declaration of four named constants:

- TAB_1
- TAB_2
- TAB_3
- TAB_4
- BACKTAB_5

Each of these five items defines an offset from the left or the right edge of the window. TABs are positive offsets measured from the left edge; BACKTABS are negative offsets measured from the right edge. The first pair help position the display of the integer registers; the second pair help position the display of the floating point registers; the fifth item positions the command buttons.

I chose to implement my graphical display as a derived class of FLTK, specifically the class **Fl_Window**. The individual widgets comprising the display window will appear as members within this new class. These elements will also be declared using FLTK components.

We will try to help clarify the use of each widget as we go along. Each of the following subsections focus on the specific choices I made for widgets at each step.

command buttons

The six command buttons seemed to me to be the best starting point. They will all be at the right edge of the display; they are just simple rectangles; and they are are at present just for display purposes (no call-back mechanism) at this point in time.

FLTK provides the widget class **Fl_Button** which is ideally suited for this component!

Our initial declarations for the **kwindow_type** class include six members: step_button, stop_button, run_button, memory_button, stack_button, and heap_button each of which is declared as an Fl_Button.

Recall that widgets are basically defined and then just sit around waiting until they are called upon to actually do something. For each command button, we will also create an additional boolean member "<name>_pushed" initialized to false to indicate the button is "waiting."

At this point it might be advantageous to look ahead for just a moment into the **kwindow** header file. The header file contains the basic structure and declarations for the **kwindow_type**. Most of the details for the actual implementation are found elsewhere in a sister "implementation" file. We will focus on implementation shortly, but I want to highlight one facet of the header file right now. Typically, any widget which will utilize a call-back mechanism must declare three supporting methods. These will be found toward the bottom of the header file.

- void wait_for_<name>_button (void);
- static void cb_<name> (Fl_Widget* w,void* d);
- void do_<name> (void);

Even though we need not concern ourselves at this time with any details regarding implementation, let me say that the first two methods are essentially boiler plate templates that universally should work.

The third method should not be boiler plate; it is specific to the task the button activation should perform. It may be trivial; it may be exceedingly complicated; but it is the response to the user's directive.

Lastly, the second method is the definition of the call-back mechanism. Recall that the source is a low-level device driver notification that an event has occurred. The device driver does not understand classes and methods; so it is not communicating in C++. The keyword **static** and the data type **void*** are essential in this declaration!

- static defines that method (function) not to a specific object but rather to the class itself
- void* is an address or pointer to some data of unspecified type

The sole purpose for the second method is to convert a basic function call with an unspecified data type into the correct method for a specific object in C++ context. It is a one-liner and can be implemented directly in the header file.

```
((kwindow_type*) d) → do_<name> ();
```

Once the command buttons have been displayed properly in its window, we can later return to the actual implementation of wait_for, call-back, and respond. I want to build the interface first before getting the bells and whistles to work properly.

integer registers

The integer register display will be comprised of three elements:

- a descriptive label for each register
- its integer decimal value as a hexadecimal bit pattern
- a descriptive label for the register set information

The register label is the standard sequence: I0, I1, ... , I14, I15. However, most registers have a pre-ordained purpose which is highlighted by its alias: IA, IB, ... , RP, HP. The descriptive label for a register will be both its standard label and its alias (if one is used).

All three of the above elements can easily be implemented using an FLTK widget called **Fl_Box**, which is especially useful for displaying character strings.

*A possible future enhancement to **visual_kbox** would be the implementation of a display option selector: hexadecimal bit pattern, unsigned integer, signed integer.*

floating point registers

Similarly, the floating point register display will be comprised of three elements:

- a descriptive label for each register
- its floating point decimal value as a hexadecimal bit pattern
- a descriptive label for the register set information

The register label is the standard sequence: F0, F1, ... , F14, F15. However, some registers have a pre-ordained purpose which is highlighted by its alias:FA, FB, FC, FD. The descriptive label for a register will be both its standard label and its alias (if one is used).

All three of the above elements can easily be implemented using an FLTK widget called **Fl_Box**, which is especially useful for displaying character strings.

*A possible future enhancement to **visual_kbox** would be the implementation of a display option selector: hexadecimal bit pattern, fixed point, scientific notation.*

program counter and flags

In the middle of the graphical user interface window, I would like to highlight:

- the program counter: a descriptive label and the counter value
- the four components of an instruction:
  opcode, operand1, operand2, operand3
- the flags: EQ, GT, LT

The descriptive label and the four instruction components are character strings. The counter value, like integer register data earlier, will be displayed in hexadecimal. Hence, all six can be implemented using the FLTK widget called **Fl_Box** .

That leaves us with the three flags EQ, GT, and LT. For these three items I chose to use the FLTK widget called **Fl_Light_Button**

console display

The console display had the appearance of being a truly complicated element of the display. Fortunately, FLTK provides a very useful widgets for this purpose called **Fl_Simple_Terminal**. We will discuss the features of this widget and all the other preceding widgets in greater detail shortly.

At this point, though, I would like to interrupt the exposition and allow the reader to review the source code I have written for the **kwindow** header file. All of the members and methods have been described. The static functions required to implement the call-back mechanism have been highlighted and explained. Hopefully, most if not all of the elements in the header file will make some sort of sense to the reader.

When we return from reading the header file source code we will resume our conversation on how to implement the required call-back mechanism elements.

4.4 kwindow.h

kwindow.h source code

```
#ifndef _KWINDOW_H
#define _KWINDOW_H

#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Output.H>                   // for Fl_Input!
#include <FL/Fl_Light_Button.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Simple_Terminal.H>

#include "kbox.h"

using namespace std;

/* ------------------------------------------------------------------ */

const int WINDOW_WIDTH  = 1600;
const int WINDOW_HEIGHT = 1000;
const int TAB1          = 25;
const int TAB2          = 225;
const int TAB3          = 1000;
const int TAB4          = 1200;
const int BACKTAB5      = -125;

static string IREG_LABELS[16]
{ "I0          IA","I1          IB",
  "I2          IC","I3          ID",
  "I4         SAR","I5         DAR",
  "I6          OR","I7          IR",
  "I8","I9","I10","I11",
  "I12         SP","I13         FP",
  "I14         RP","I15         HP" };

static string FREG_LABELS[16]
{ "F0          FA","F1          FB",
  "F2          FC","F3          FD",
  "F4","F5","F6","F7",
  "F8","F9","F10","F11",
  "F12","F13","F14","F15" };

/* ------------------------------------------------------------------ */

class kwindow_type : public Fl_Window
{
public:
  Fl_Button*          step_button;
  bool                step_button_pushed;
  Fl_Button*          stop_button;
  bool                stop_button_pushed;

  Fl_Button*          run_button;
  bool                run_button_pushed;

  Fl_Button*          memory_button;
  bool                memory_button_pushed;

  Fl_Button*          stack_button;
  bool                stack_button_pushed;

  Fl_Button*          heap_button;
  bool                heap_button_pushed;

  Fl_Box*             i_registers_label;
  Fl_Box*             i_reg_names[16];
  Fl_Box*             i_reg_values[16];

  Fl_Box*             f_registers_label;
  Fl_Box*             f_reg_names[16];
  Fl_Box*             f_reg_values[16];

  Fl_Box*             pc_label;
  Fl_Box*             pc_value;

  Fl_Box*             pc_opcode;
  Fl_Box*             pc_oper1;
  Fl_Box*             pc_oper2;
  Fl_Box*             pc_oper3;

  Fl_Light_Button*    eq_flag;
  Fl_Light_Button*    gt_flag;
  Fl_Light_Button*    lt_flag;

  Fl_Simple_Terminal* console_display;

/* ------------------------------------------------------------- */

  kwindow_type (void);

/* ------------------------------------------------------------- */

  void wait_for_step_button (void);

  static void cb_step (Fl_Widget* w,void* d)
  {   ((kwindow_type*)d)->do_step();   }

  void do_step (void);

/* ------------------------------------------------------------- */
```

```
  void wait_for_stop_button (void);

  static void cb_stop (Fl_Widget* w,void* d)
  {  ((kwindow_type*)d)->do_stop();  };

  void do_stop (void);

/* ------------------------------------------------------------------ */

  void wait_for_run_button (void);

  static void cb_run (Fl_Widget* w,void* d)
  {  ((kwindow_type*)d)->do_run();  };

  void do_run (void);

/* ------------------------------------------------------------------ */

  void wait_for_memory_button (void);

  static void cb_memory (Fl_Widget* w,void* d)
  {  ((kwindow_type*)d)->do_memory();  };

  void do_memory (void);

/* ------------------------------------------------------------------ */

  void wait_for_stack_button (void);

  static void cb_stack (Fl_Widget* w,void* d)
  {  ((kwindow_type*)d)->do_stack(); };

  void do_stack (void);

/* ------------------------------------------------------------------ */


  void wait_for_heap_button (void);

  static void cb_heap (Fl_Widget* w,void* d)
  {  ((kwindow_type*)d)->do_heap(); };

  void do_heap (void);

/* ------------------------------------------------------------------ */

  void fedex (klunk,bool&);

  void refresh ();
};

/* ------------------------------------------------------------------ */
```

```
void popup (string prompt,char* buffer);

/* ----------------------------------------------------------------------- */

#endif // _KWINDOW_H
```

4.5 KBOX Display (Part II)

As we return to implementing our graphical user interface, we continue discussing the members and the methods for the **kwindow_type** class.

Remember our class is a derived class of **Fl_Window** and our first concern at this time is defining a default constructor.

The constructor for **Fl_Window** requires five arguments be specified: (x,y,w,h,label) with integers x and y being offsets from the upper left hand corner of the screen, integer w being the window width, integer h being the window height, and a cstring label being the value to be displayed at the top of the window.

Our first six components are command buttons. Each of these will be represented by an **Fl_Button** with an addition boolean member to indicate the button is waiting or has been activated.

The constructor for **Fl_Button** also requires five arguments be specified: (x,y,w,h,label). The cstring label in this argument list will appear centered within the button (an ampersand & in front of a character makes that character a shortcut key for the button).

For all six command buttons I set the width to 100 and the height to 50. **Fl_Button** also provides several methods for customizing the appearance of the button:

- color (<color>)
  sets the background
- labelcolor (<color>)
  sets the foreground
- align (<align>)
  set the label alignment
- copy_label(<new_label>)
  update or modify the button label
- callback (<name>,(void*) this)
  defines the call-back mechanism

This information regarding **Fl_Button** is certainly sufficient to understand the initial C++ source code implementation of the command buttons within the **kwindow_type** class constructor.

We can now move on to the the display of the register sets – both the integer register set and the floating point register set. Each display is constructed using three FLTK **Fl_Boxes**.

The constructor for **Fl_Box** requires six arguments be specified: (type,x,y,w,h,label) with type indicating a specific visual effect for the box (FL_UP_BOX).

For the actual register names I set the width to 200 and the height to 40; for the register values I set the width to 250 and the height to 40; for the top label display box I set the width to 450 and the height to 50. This is true for both the integer register set and the floating point register set.

Fl_Box also provides several methods for customizing the appearance of the button:

- color (<color>)
  sets the background
- labelcolor (<color>)
  sets the foreground
- copy_label(<new_label>)
  update or modify the button label
- align (<align>)
  sets the alignment

Moving on to the display of the **kbox** flags, we next implement theses three indicators using the FLTK **Fl_Light_Button**. The **Fl_Light_Button** has a label and a light indicator (ON for true and OFF for false).

The constructor for **Fl_Light_Button** requires five arguments be specified: (x,y,w,h,label).

Fl_Light_Button also provides a method of changing the status of the indicator:

- value (tf)
  update or modify the button indicator
  note: our source code (not the indicators)
  must check for valid settings

We have now completed most of the heavy lifting regarding the definition of our constructor for the **kwindow_type**. But there remains one last feature that we must consider.

At the very bottom of our graphical user interface, we want a console display to show the results of executing our **kcode** instructions. It must prompt for input according to the source code; it must display the input in the console exactly as if it were transferred directly from the keyboard (more on this shortly!). It will also be used to display any results of selecting **step**, **exit** and **run** when these command buttons are pressed.

The very good news is that FLTK provides a very useful widget expressly for this purpose – **Fl_Simple_Terminal**.

The constructor for **Fl_Simple_Terminal** requires four arguments be specified: (x,y,w,h).

Fl_Simple_Terminal also provides a several important methods:

- printf (<output_string>)
  typical C printf output expression
- ansi (tf)
  enable / disable recognition of ANSI sequences
  which change font color, face, and size

As we come to the end of our presentation regarding the constructor for a **kwindow_type** (which is a derived class from FLTK **Fl_Window**) we return to two additional methods for the class:

- resizable (this)
  which makes the derived class resizable
  i.e., the window may be resized,
  together with its contents

- set_modal ()
  will remain on top of other windows
  will prevent events from being delivered to other windows

Setting our **kwindow_type** to be resizable is certainly a good thing to do. Different users of the software may have different preferences. Different display devices may present the image in slightly differing ways. This attribute will allow a minimal level of control over what the individual might prefer.

We will discuss the set_modal method after we have completed our presentation regarding the implementation of the class **kwindow_type**.

4.6 kwindow.cpp

kwindow.cpp source code

```
#include <cstdlib>
#include <iostream>
#include <cstring>

#include "kwindow.h"

using namespace std;

extern kbox_type        VISUAL_KBOX;
extern klunk            MAXPC;

/* ---------------------------------------------------------------- */

kwindow_type::kwindow_type (void)
: Fl_Window
    { 0,0,WINDOW_WIDTH,WINDOW_HEIGHT,"VISUAL KBOX" }

{
  //basic control buttons for the KBOX simulator
  {
    step_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,25,100,50,"&STEP" };
    step_button->color(FL_GREEN);
    step_button->labelcolor(FL_BLACK);
    step_button->align(FL_ALIGN_CENTER);
    step_button->callback(cb_step,(void*)this);
    step_button_pushed = false;

    stop_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,85,100,50,"E&XIT" };
    stop_button->color(FL_RED);
    stop_button->labelcolor(FL_BLACK);
    stop_button->align(FL_ALIGN_CENTER);
    stop_button->callback(cb_stop,(void*)this);
    stop_button_pushed = false;

    run_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,145,100,50,"&RUN " };
    run_button->color(FL_YELLOW);
    run_button->labelcolor(FL_BLACK);
    run_button->align(FL_ALIGN_CENTER);
    run_button->callback(cb_run,(void*)this);
    run_button_pushed = false;

    memory_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,250,100,50,"&MEMORY" };
    memory_button->color(FL_BLUE);
    memory_button->labelcolor(FL_WHITE);
    memory_button->align(FL_ALIGN_CENTER);
    memory_button->callback(cb_memory,(void*)this);
    memory_button_pushed = false;

    stack_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,310,100, 50,"S&TACK " };
    stack_button->color(FL_BLUE);
    stack_button->labelcolor(FL_WHITE);
    stack_button->align(FL_ALIGN_CENTER);
    stack_button->callback(cb_stack,(void*)this);
    stack_button_pushed = false;

    heap_button = new Fl_Button
      { WINDOW_WIDTH+BACKTAB5,370,100,50,"&HEAP  " };
    heap_button->color(FL_BLUE);
    heap_button->labelcolor(FL_WHITE);
    heap_button->align(FL_ALIGN_CENTER);
    heap_button->callback(cb_heap,(void*)this);
    heap_button_pushed = false;
  }

  //integer register display
  {
    i_registers_label = new Fl_Box
      { FL_UP_BOX,TAB1,25,450,50,
      "INTEGER REGISTERS" };
    i_registers_label->color(FL_WHITE);
    i_registers_label->labelcolor(FL_RED);
    for (int i=0;i<16;i++)
    {
      string i_reg = "I"+to_string(i);
      i_reg_names[i] = new Fl_Box
        { FL_UP_BOX,TAB1,75+i*40,200,40,
        IREG_LABELS[i].data() };
      i_reg_names[i]->color(FL_WHITE);
      i_reg_names[i]->labelcolor(FL_RED);
      i_reg_names[i]->align(FL_ALIGN_CENTER);
      i_reg_values[i] = new Fl_Box
        { FL_UP_BOX,TAB2,75+i*40,250,40,klunk2hex
        (VISUAL_KBOX.REGISTERS.get_register(i_reg)) };
      i_reg_values[i]->color(FL_WHITE);
      i_reg_values[i]->labelcolor(FL_BLACK);
      i_reg_values[i]->align(FL_ALIGN_CENTER);
    }
  }

  //floating point register display
  {
    f_registers_label = new Fl_Box
      { FL_UP_BOX,TAB3, 25,450, 50,
      "FLOATING POINT REGISTERS" },
    f_registers_label->color(FL_WHITE);
    f_registers_label->labelcolor(FL_RED);
    for (int i=0;i<16;i++)
    {
      string f_reg = "F"+to_string(i);
      f_reg_names[i] = new Fl_Box
        { FL_UP_BOX,TAB3,75+i*40,200,40,
        FREG_LABELS[i].data() };
      f_reg_names[i]->color(FL_WHITE);
      f_reg_names[i]->labelcolor(FL_RED);
      f_reg_names[i]->align(FL_ALIGN_CENTER);
      f_reg_values[i] = new Fl_Box
        { FL_UP_BOX,TAB4,75+i*40,250,40,klunk2hex
        (VISUAL_KBOX.REGISTERS.get_register(f_reg)) };
      f_reg_values[i]->color(FL_WHITE);
      f_reg_values[i]->labelcolor(FL_BLACK);
      f_reg_values[i]->align(FL_ALIGN_CENTER);
    }
  }

  //program counter information
  {
    pc_label = new Fl_Box
      { FL_UP_BOX,500,150,200,50,
      "PROGRAM_COUNTER" };
    pc_label->color(FL_WHITE);
    pc_label->labelcolor(FL_RED);
    pc_label->align(FL_ALIGN_CENTER);

    pc_value = new Fl_Box
      { FL_UP_BOX,725,150,250,50,
      klunk2hex(VISUAL_KBOX.FLAGS.getPC()) };
    pc_value->color(FL_WHITE);
    pc_value->labelcolor(FL_BLACK);
    pc_value->align(FL_ALIGN_CENTER);
  }

  //instruction display
  {
    if (MAXPC == 0)
    {
      cout << " ... that's all folks!\n";
      exit(EXIT_SUCCESS);
    }
    code_entry instruction =
      VISUAL_KBOX.INSTRUCTIONS.
      get_instruction(0);
/*
Please note the variation in coding that follows:

The label in the Fl_Box constructor is "".
The actual label is defined using the copy_label method,
which is typically used to modify/update during runtime.

For some unknown reason (at least to me!), when I
```

```
compile and run the code with the value in the Fl_Box
constructor, the initial display contains gibberish.
Using the copy_label method the display is fine.

I find that the distinction between C++ string class and
C strings (char* and char[] especially present when
working with C++ and FLTK.
*/
    pc_opcode = new Fl_Box
      { FL_UP_BOX,640,260,200,50,"" };
    pc_opcode->color(FL_WHITE);
    pc_opcode->labelcolor(FL_BLACK);
    pc_opcode->align(FL_ALIGN_CENTER);
    pc_opcode->copy_label
      (instruction.get_opcode().data());

    pc_oper1 = new Fl_Box
      { FL_UP_BOX,640,320,200,50,"" };
    pc_oper1->color(FL_WHITE);
    pc_oper1->labelcolor(FL_BLACK);
    pc_oper1->align(FL_ALIGN_CENTER);
    pc_oper1->copy_label
      (instruction.get_operand1().data());

    pc_oper2 = new Fl_Box
      { FL_UP_BOX,600,380,280,50,"" };
    pc_oper2->color(FL_WHITE);
    pc_oper2->labelcolor(FL_BLACK);
    pc_oper2->align(FL_ALIGN_CENTER);
    pc_oper2->copy_label
      (instruction.get_operand2().data());

    pc_oper3 = new Fl_Box
      { FL_UP_BOX,640,440,200,50,"" };
    pc_oper3->color(FL_WHITE);
    pc_oper3->labelcolor(FL_BLACK);
    pc_oper3->align(FL_ALIGN_CENTER);
    pc_oper3->copy_label
      (instruction.get_operand3().data());
  }

  //comparison flags display
  {
    eq_flag = new Fl_Light_Button
      { 600,550,50,50,"EQ" };
    eq_flag->value(VISUAL_KBOX.FLAGS.getEQ());
    gt_flag = new Fl_Light_Button
      { 712,550,50,50,"GT" };
    gt_flag->value(VISUAL_KBOX.FLAGS.getGT());
    lt_flag = new Fl_Light_Button
      { 825,550,50,50,"LT" },
    lt_flag->value(VISUAL_KBOX.FLAGS.getLT());
  }
  //console display for to show program execution
  {
    console_display = new Fl_Simple_Terminal
      { 0,740,WINDOW_WIDTH,260 };
    console_display->ansi(true);
  }

  resizable(this);
  end();
}

/* ------------------------------------------------------------------ */

void kwindow_type::wait_for_step_button (void)
{
  while (!step_button_pushed)
    Fl::wait();
  step_button_pushed = false;
  redraw();
}

void kwindow_type::do_step (void)
{
  klunk pc = VISUAL_KBOX.FLAGS.getPC();
  bool halt_flag;
  fedex(pc,halt_flag);
  refresh();
}

/* ------------------------------------------------------------------ */

void kwindow_type::wait_for_stop_button (void)
{
  while (!stop_button_pushed)
    Fl::wait();
  stop_button_pushed = false;
  redraw();
}

void kwindow_type::do_stop (void)
{
  console_display->printf(" ... that's all folks!\n");
  exit(EXIT_SUCCESS);
}

/* ------------------------------------------------------------------ */

void kwindow_type::wait_for_run_button (void)
{
  while (!run_button_pushed)
    Fl::wait();
  run_button_pushed = false;
  redraw();
}

void kwindow_type::do_run (void)
{
  klunk pc = VISUAL_KBOX.FLAGS.getPC();
  bool halt_flag = false;
  while (true)
  {
    fedex(pc,halt_flag);
    refresh();
    if (halt_flag)
      break;
    pc = VISUAL_KBOX.FLAGS.getPC();
  }
}

/* ------------------------------------------------------------------------- */

void kwindow_type::wait_for_memory_button (void)
{
  while (!memory_button_pushed)
    Fl::wait();
  memory_button_pushed = false;
  redraw();
}

void kwindow_type::do_memory (void)
{
  console_display->printf
    ("GLOBAL/STATIC MEMORY MAP:\n");
  for (klunk i=0;i<VISUAL_KBOX.MEMORY.map.size();i++)
    console_display->printf("%s  %d  %d\n",
      VISUAL_KBOX.MEMORY.map[i].get_name().data(),
      VISUAL_KBOX.MEMORY.map[i].get_address(),
      VISUAL_KBOX.MEMORY.map[i].get_size());
  console_display->printf("\n");
  console_display->printf
    ("GLOBAL/STATIC MEMORY:\n");
  for (klunk i=0;
       i<VISUAL_KBOX.MEMORY.get_next_available();
       i++)
    console_display->printf
      ("%d  %s\n",i,
      klunk2hex(VISUAL_KBOX.MEMORY.peek(i)));
  console_display->printf("\n");
}

/* ------------------------------------------------------------------------- */

void kwindow_type::wait_for_stack_button (void)
{
  while (!stack_button_pushed)
```

```
    Fl::wait();
  stack_button_pushed = false;
  redraw();
}

void kwindow_type::do_stack (void)
{
  console_display->printf
    ("STACK:\n");
  string sp_reg = "I12";
  klunk sp = VISUAL_KBOX.REGISTERS.get_register(sp_reg);
  if (sp == MEMORY_SIZE)
    console_display->printf
      ("<empty>\n");
  else
    for (klunk i=MEMORY_SIZE-1;i>=sp;i--)
      console_display->printf
        ("%d  %s\n",
        i,klunk2hex(VISUAL_KBOX.MEMORY.peek(i)));
  console_display->printf("\n");
}

/* ------------------------------------------------------------ */

void kwindow_type::wait_for_heap_button (void)
{
  while (!heap_button_pushed)
    Fl::wait();
  heap_button_pushed = false;
  redraw();
}

void kwindow_type::do_heap (void)
{
  console_display->printf
    ("HEAP:\n");
  string hp_reg = "I15";
  klunk hp = VISUAL_KBOX.REGISTERS.get_register(hp_reg);
  if (hp == VISUAL_KBOX.MEMORY.get_next_available())
    console_display->printf
      ("<empty>\n");
  else
    for (klunk i=VISUAL_KBOX.MEMORY.get_next_available();
        i<hp;
        i++)
      console_display->printf
        ("%d  %s\n",
        i,klunk2hex(VISUAL_KBOX.MEMORY.peek(i)));
  console_display->printf("\n");
}

/* ------------------------------------------------------------ */
```

```
/*
void kwindow_type::fedex (klunk pc,bool& halt_flag)
is a very large implementation and
may be found in a separate source fie:

     kfedex.cpp

*/

/* ------------------------------------------------------------------ */

void popup (string prompt,char buffer[])
{
  Fl_Window window(400,120,prompt.data());
  window.set_modal();
  Fl_Input input_data(100,20,200,40);
  Fl_Button okay_btn(100,75,75,25,"&OK");
  Fl_Button cancel_btn(225,75,75,25,"&CANCEL");
  window.end();
  window.show();
  while (true)
  {
    Fl::wait();
    Fl_Widget *o;
    while (o = Fl::readqueue())
      if (o == &okay_btn)
      {
        strcpy(buffer,input_data.value());
        return;
      }
      else if (o == &cancel_btn || o == &window)
      {
        strcpy(buffer,"\0");
        return;
      }
  }
}

/* ------------------------------------------------------------------ */

void kwindow_type::refresh (void)
{
  for (klunk i=0;i<16;i++)
  {
    string i_reg = "I"+to_string(i);
    string f_reg = "F"+to_string(i);
    i_reg_values[i]->copy_label
      (klunk2hex
      (VISUAL_KBOX.REGISTERS.get_register(i_reg)));
    f_reg_values[i]->copy_label
      (klunk2hex
      (VISUAL_KBOX.REGISTERS.get_register(f_reg)));
  }
  klunk pc = VISUAL_KBOX.FLAGS.getPC();
  pc_value->copy_label(klunk2hex(pc));
  if (pc < MAXPC)
  {
    code_entry next_instr =
      VISUAL_KBOX.INSTRUCTIONS.get_instruction(pc);
    pc_opcode->copy_label
      (next_instr.get_opcode().data());
    pc_oper1->copy_label
      (next_instr.get_operand1().data());
    pc_oper2->copy_label
      (next_instr.get_operand2().data());
    pc_oper3->copy_label
      (next_instr.get_operand3().data());
    eq_flag->value(VISUAL_KBOX.FLAGS.getEQ());
    gt_flag->value(VISUAL_KBOX.FLAGS.getGT());
    lt_flag->value(VISUAL_KBOX.FLAGS.getLT());
  }
  else
  {
    pc_opcode->copy_label("HALT");
    pc_oper1->copy_label("");
    pc_oper2->copy_label("");
    pc_oper3->copy_label("");
    eq_flag->value(true);
    gt_flag->value(false);
    lt_flag->value(false);
  }
}
```

4.7 KBOX Display (Part III)

In this section we tie up any loosends that remain. Three important items still remain:

- implementation of the call-back mechanism
- implementation of **kbox** input from the user
- syncing **kbox** data with **kwindows** data
  before redrawing the window

4.7.1 call-back mechanism

Each button has three components for the call-back mechanism:

- wait-for
- system call-back
- do something

The following C++ template provides suitable code for the wait-for implementation:

```
while ( !<button>_pushed )         // wait loop
   Fl::wait ();
<button>_pushed = false;           // reset flag
Fl::redraw ();                     // redraw window
```

We have already implemented the system call-back in the header file, but we remind the reader here of the basic element:

```
( ( kwindow_type* ) d) -> do_<button> ();
```

We convert the system call-back into the correct derived class member and then activate its method for handling the event.

The real work now lies ahead in doing what is requested by the user. This phase may be short and simple; or it may be long and difficult. It all depends on what the the button represents and how it is expected to perform. We will discuss in turn each of our six command buttons.

step button

If the step button is pressed, we expect **visual_kbox** to execute the instruction displayed in the center of the display screen. Depending on the specific instruction, the instruction may retrieve or store information in memory or it may calculate or compare items in registers, but at the very end it must update the flags to the correct values.

Kfedex will be the largest implementation file we will encounter in **visual_kbox**; yet it will contain only a single method **fedex** definition. Each and every **kcode** instruction must be correctly implemented in this code.

We postpone discussion regarding this file until later in the text, but suffice it to say now that all the work has been done before in the original **kbox** simulator. The boolean variable halt_flag is a return indicator from the **fedex** method that a halt instruction has been encountered.

refresh will be discussed at the end of this section. The purpose for this function is specifically to synchronize the data found in the **kbox** simulator with the data displayed in the **visual_kbox** window.

stop button

If the stop button is pressed, we expect **visual_kbox** to terminate execution. Its implementation has two components: writing a termination message to the console display and terminating the C++ program.

Please note:

The **kbox halt** instruction does **not** terminate the assembly language simulator; it does not execute any instruction; it does not advance the program counter. It simply announces that the program is at a **halt** instruction.

The **kbox nop** instruction also does not execute any instruction; but it does advance the program counter. I truly does nothing but take up space!

The **stop button** is the correct vehicle to leave the **visual_kbox** graphical user interface.

run button

If the run button is pressed, we expect **visual_kbox** to execute the entire instruction set from the current position (program counter) until the source code ends (an *implicit* **halt**) or a **halt** instruction (an *explicit* **halt**).

The implementation is exceedingly straightforward – we simply incorporate the **fedex** method into a looping structure until a **halt** instruction is executed.

memory button

The coding for the memory button, as well as the following two buttons, is exceedingly simple. We need to show the contents of static/global memory in the console display.

This is a simple counted loop displaying the contents in memory from address 0 up to but not including address next_available.

In addition, pressing the memory button will also display the information found in the memory map: the name of the identifier, its storage address in memory, and the storage size (in klunks) .

stack button

Recall that the stack grows downward in memory. This is also a simple counted loop displaying the contents in memory from address MEMORY_SIZE-1 down to and including the address found in the stack pointer register (SP).

heap button

Recall that the heap grows upward in memory. This is also a simple counted loop displaying the contents in memory from address next_available up to but not including the address found in the heap pointer register (HP).

4.7.2 visual_kbox input

Our console display (at the very bottom of our graphical user interface) is implemented as an Fl_Simple_Terminal. The graphical display has the appearance of a typical command line interface. However, the console display is limited to displaying output only. In order to perform input operations, we need to use a second Fl_Window.

This window will be a popup window: it will appear when necessary and it will disappear when the job is done.

By now the following code for the popup menu should be easily understood:

- the window contains three elements:
  - an input box (Fl_Input) called input_data
  - an okay button (Fl_Button) called okay_btn
  - a cancel button (Fl_Button) called cancel_btn
- this popup box uses **set_modal**
  which places the box on top
  directs all events to the popup box
- the readqueue method is an alternate form of call-back mechanism in FLTK

I found this piece of code on the Internet as I was nearing the end of my project. It is a self-contained, very basic implementation that uses the default call-back mechanism in FLTK, called **readqueue**.

Although this popup window performs the actual data transfer for input into the **visual_kbox** simulation, we still want the console display to echo our input! This will be done as part of the **kfedex** implementation. The cstring buffer that returns from the popup window will be written to the console display as part of the source code handling the **kbox** GET instruction.

4.7.3 synchronized data

This project has been built on two key elements:

- the underlying kbox simulator has its own data and methods
- the visually pleasing visual_kbox interface has its own data and methods

Much of the coding that remains will require us to be aware that the two underlying elements are separate but that the two underlying elements are also closely related. Modifying a comparison flag in the simulator does not modify the flag in the graphical display; and vice versa. Any time a value is modified, the programmer must ensure that they are consistent or synchronized.

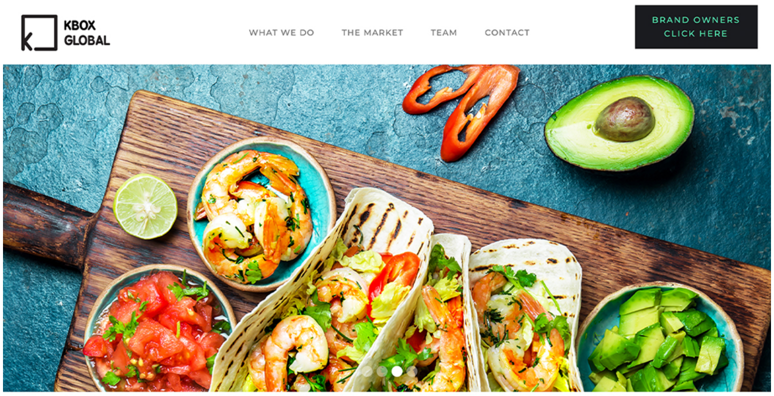
Since this is my initial foray into graphical programming. I do not want to give a definitive recommendation on the best way to handle this issue. But, as a beginner approaching a new style of programming, I chose to focus on having the underlying data well-organized and accurate and then updating all the data in the graphical display before redrawing the window.

My source code for synchronizing the graphical display with the underlying simulator data is called **refresh**. It is the last item in the **kwindows** implementation file. It is exceedingly redundant in that it synchronizes everything whether it needs to or not! A possible alternative might be to synchronize specifically for each instruction. This would probably prove to be slightly faster than a universal refresh method. But the **kfedex** file is already humongous and would require writing customized synchronization for each instruction in **kcode**.

Chapter 5

KSETUP Implementation

KBOX Food Supplier!



5.1 Overview

At this point in time only two additional implementation files for our **visual_kbox** simulator need to be clarified. However, both files are fairly large in size. The first file, **ksetup**, oversees the processing of the **kcode** assembly language file and setting up the simulator to begin executing the instruction set.

Processing includes several different aspects:

- reading the source file while simultaneously performing lexical analysis on its contents
- collecting information that is pertinent to the **_DATA_SEGMENT**
  _DEFINE directives
  _RESERVE directives
- collecting information that is pertinent to the **_CODE_SEGMENT**
  _GLOBAL directives
  _EXTERN directives
  _LABEL directives
  kcode instructions
- initialize global/static memory, setting key registers, and initializing flags

In the following pages we will discuss each of the items.

5.2 lexical analysis

In the book **Fun With Programming Languages** we saw that the first step in analyzing and understanding a language was to identify its building block components. So the role of this C++ component is to open the source file and to process each line, ignoring the blank lines and the comment lines, identifying whether the line is a directive or an instruction, and identifying the required operands which follow.

The header file **ksource.h** declares the methods that will be defined in the implementation file **ksource.cpp**.

- open (name)
  opens the **kcode** source file for input
- close ()
  closes the source file
- get_opcode ()
  return the directive or the instruction
    a directive starts with an undscore (_)
    an instruction does not
- get_operand ()
  address, register, immediate value
- read_line ()
  skipping blank and/or comment lines
- trim_lead_blanks ()
  ignore leading white space
- is_eol ()
- is_eof ()

The implementation file should contain no surprises. get_opcode is very simple; get_operand is more complicated only because there are more options to consider:

- identifiers (labels)
  initial character is alphabetic
- character literal
  initial character is single quote (')
- string literal
  initial character is double quote (")
- numeric literal (integer)
  initial character is +, -, or digit
- numeric literal (fixed point real)
  sequence contains a period (.)
- numeric literal (scientific notation real)
  sequence contains E or e followed by integer
- memory address
  initial character is equal sign (=)

An end of line marker (#) is appended to every input line from the source code file to aid in the lexical analysis. Specifically the marker is intended to help identify blank lines as well as ignore comments to the right of directives and instructions.

The three methods trim_lead_blanks (), is_eol (), and is_eof () are implemented to assist in the lexical analysis by identifying and removing white space.

My source code for **ksource** follows. After that we look at the three components necessary to prepare the simulator for operation:

- processing the data_segment
- processing the code_segment
- initializing static/global memory, important registers, and **kbox** flags

5.3 ksource

ksource.h

```
#ifndef _KSOURCE_H
#define _KSOURCE_H

#include <cstdlib>
#include <fstream>

using namespace std;

/* ----------------------------------------------------------------------------- */

class ksource_type
{
private:
  fstream       kfile;          // source file
  string        buffer;         // input buffer
  int           pos;            // current position
  bool          done;           // file has been read

public:
  void open (const string&);
  void close (void);

  string get_opcode (void);
  string get_operand (void);

  bool read_line (void);

  void trim_lead_blanks (void);

  bool is_eol (void) const;
  bool is_eof (void) const;
};

/* ----------------------------------------------------------------------------- */

#endif // _KSOURCE_H
```

ksource.cpp

```
#include <cstdlib>
#include <string>
#include <cstring>
#include <iostream>
#include <fstream>

#include "ksource.h"

using namespace std;

/* ---------------------------------------------------------------- */

void ksource_type::open (const string& name)
{
  kfile.open(name);
  if (kfile.fail())
  {
    cerr << "source file (" << name
         << ") not found!" << endl;
    exit(EXIT_FAILURE);
  }
  else
    done = false;
}

void ksource_type::close (void)
{
  kfile.close();
}

string ksource_type::get_opcode (void)
{
  string result("");
  int i = 0;
  if (buffer[pos] == '_')
  {
  // KCODE directive
    while (isalpha(buffer[pos]) ||
           (buffer[pos] == '_'))
      result += toupper(buffer[pos++]);
    trim_lead_blanks();
  }
  else if (isalpha(buffer[pos]))
  {
  // KCODE instruction
    while (isalnum(buffer[pos]))
      result += toupper(buffer[pos++]);
    trim_lead_blanks();
  }
  else
  {
```

```
  // invalid KCODE entry
    cerr << "invalid line: " << buffer << endl;
    exit (EXIT_FAILURE);
  }
  return result;
}

string ksource_type::get_operand (void)
{
  string result("");
  int i = 0;
  if (is_eol())
    return result;
  else if (isalpha(buffer[pos]))
  {
  // label or identifier
    while (isalnum(buffer[pos]))
      result += toupper(buffer[pos++]);
    trim_lead_blanks();
  }
  else if ((buffer[pos] == '\"') ||
           (buffer[pos] == '\''))
  {
    // character or string literal
    char terminal = buffer[pos];
    result += buffer[pos++];
    while (buffer[pos] != terminal)
      result += buffer[pos++];
    result += buffer[pos++];
    trim_lead_blanks();
  }
  else if ((buffer[pos] == '+') ||
           (buffer[pos] == '-') ||
           (isdigit(buffer[pos])))
  {
    // numeric literal: integer
    if ((buffer[pos] == '+') || (buffer[pos] == '-'))
      result += buffer[pos++];
    while (isdigit(buffer[pos]))
      result += buffer[pos++];
    // numeric literal: floating point
    if (buffer[pos] == '.')
    {
      result += buffer[pos++];
      while (isdigit(buffer[pos]))
        result += buffer[pos++];
    }
    // numeric literal: floating point
    if ((buffer[pos] == 'e') || (buffer[pos] == 'E'))
    {
      result += buffer[pos++];
      while (isdigit(buffer[pos]))
        result += buffer[pos++];
    }
    // numeric literal: floating point
    if ((buffer[pos] == 'e') || (buffer[pos] == 'E'))
    {
      result += buffer[pos++];
      if ((buffer[pos] == '+') || (buffer[pos] == '-'))
      {
        result +=  buffer[pos++];
        while (isdigit(buffer[pos]))
          result += buffer[pos++];
      }
    }
    trim_lead_blanks();
  }
  else if (buffer[pos] == '=')
  {
  // memory address
    result += buffer[pos++];
    while (isalnum(buffer[pos]))
      result += buffer[pos++];
    trim_lead_blanks();
  }
  else
  {
    trim_lead_blanks ();
  }
  return result;
}

bool ksource_type::read_line (void)
{
  if (kfile.eof())
  {
    done = true;
    buffer = "DONE#";
    pos = 0;
    return false;
  }
  getline(kfile,buffer);
  buffer += '#';
  pos = 0;
  trim_lead_blanks();
  if (is_eol())
    return read_line();
  else
    return true;
}

void ksource_type::trim_lead_blanks (void)
{
 while ((pos < buffer.size()) &&
        (isblank(buffer[pos])))
    pos++;
```

```
}

bool ksource_type::is_eol (void) const
{
  if ((pos < buffer.size()) &&
      (buffer[pos] == '#'))
    return true;
  else
    return false;
}

bool ksource_type::is_eof (void) const
{
  return done;
}
```

5.4 _data_segment

The **_data_segment** is very simple to describe. It contains only two types of directives (which may be intermixed):

- a _DEFINE directive
- a _RESERVE directive

The _DEFINE directive serves to create an identifier and initialize it with a specific data value (exactly one klunk in size).

The _RESERVE directive serves to create an identifier and set aside a specific number of klunks but with no initialization.

The _data_segment is concluded when there are no more _DEFINE or _RESERVE directives to process.

Lastly, the _data_segment may appear either before or after the _code_segment. However, it is important to recognize that the two segments may not overlap!

5.5 _code_segment

The **_code_segment** is not quite so simple to describe. It contains three types of directives together with any valid **kcode** instruction:

- a _GLOBAL directive
- an _EXTERN directive
- a _LABEL directive

The first two directives, _GLOBAL and _EXTERN, serve no real purpose in **kcode**! Most assembly languages provide mechanisms for different components to be combined into an executable program. The _GLOBAL directive typically shares information specific to the given file with other files; the _EXTERN directive typically defines information that should be found within other files. Since our simulator is an interpreter and not a compiler, we really do not have to concern ourselves with these issues.

The first two directives may be intermixed, but may no longer be used once the first _LABEL directive or **kcode** instruction is encountered. Hence, _GLOBAL and _EXTERN directives come first; _LABEL directives and instructions come second.

This second phase is implemented in **ksetup** as its own c++ function **process_instructions (name)**.

The _code_segment is concluded when there are no more _LABEL directives or **kcode** instructions to process.

Lastly, as stated earlier, the _code_segment may appear either before or after the _data_segment. However, it is important to recognize that the two segments may not overlap!

5.6 initialization

The final component in setting up the **kbox** simulator is:

- to initialize global/static memory according to the _DEFINE and _RESERVE directives encountered in the source file,
- to initialize the stack pointer to MEMORY_SIZE,
- to initialize the frame pointer to MEMORY_SIZE,
- to initialize the return pointer to MEMORY_SIZE,
- to initialize the heap pointer to get_next_available,
- to initialize the program counter to 0, and
- to initialize the EQ flag to true (with GT and LT both false),
- to initialize MAXPC to the number of **kcode** instructions in the source file.

5.7 ksetup

ksetup.h

```
#ifndef _KSETUP_H
#define _KSETUP_H

 using namespace std;

/* ------------------------------------------------------------ */


#include "kbox.h"
#include "ksource.h"

 using namespace std;

/* ------------------------------------------------------------ */

 void setup_kbox (const string& source_file_name);
 void process_data_segment (ksource_type&);
 void process_code_segment (ksource_type&);
 void process_instructions (ksource_type&);

/* ------------------------------------------------------------ */

#endif // _KSETUP_H
```

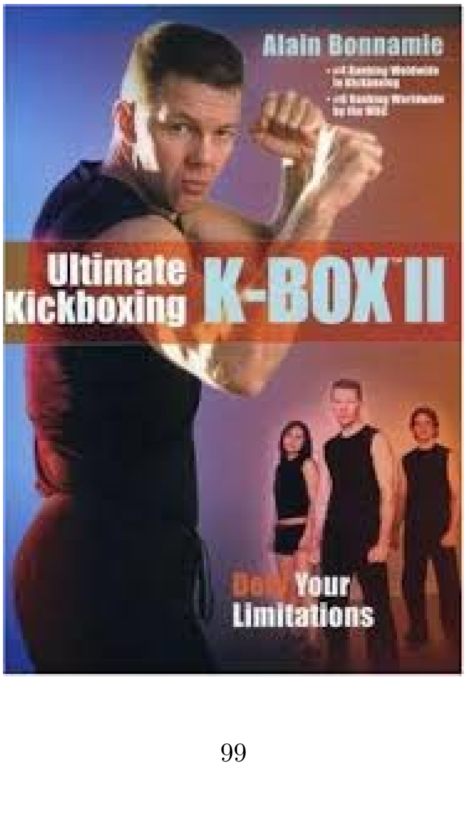
ksetup.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cstring>

#include "kbox.h"
#include "ksource.h"
#include "ksetup.h"

using namespace std;

extern kbox_type       VISUAL_KBOX;
extern ksource_type    KBOX_SOURCE;

/* ------------------------------------------------------------------ */

  long long int MAXPC;

  static bool   data_segment_started = false;
  static bool   data_segment_finished = false;
  static bool   code_segment_started = false;
  static bool   code_segment_finished = false;
  static bool   instructions_started = false;
  static bool   instructions_finished = false;

  string        item;

void setup_kbox (const string& source_file_name)
{
  ksource_type KBOX_SOURCE;
  KBOX_SOURCE.open(source_file_name);
  KBOX_SOURCE.read_line();
  if (KBOX_SOURCE.is_eof())
  {
    cerr << " ... empty source file!\n";
    exit(EXIT_FAILURE);
  }
  item = KBOX_SOURCE.get_opcode();
  if (item == "_DATA_SEGMENT")
  {
    process_data_segment(KBOX_SOURCE);
    data_segment_finished = true;
    process_code_segment(KBOX_SOURCE);
    code_segment_finished = true;
  }
  else if (item == "_CODE_SEGMENT")
  {
    process_code_segment(KBOX_SOURCE);
    code_segment_finished = true;
    process_data_segment(KBOX_SOURCE);
    data_segment_finished = true;
  }
  else
  {
    cerr << " ... unexpected item (" << item << ")!\n";
    exit(EXIT_FAILURE);
  }
  KBOX_SOURCE.close();

  string zr_reg = "I11";
  string sp_reg = "I12";
  string fp_reg = "I13";
  string rp_reg = "I14";
  string hp_reg = "I15";
  VISUAL_KBOX.REGISTERS.set_register
    (zr_reg,0);
  VISUAL_KBOX.REGISTERS.set_register
    (sp_reg,MEMORY_SIZE);
  VISUAL_KBOX.REGISTERS.set_register
    (fp_reg,MEMORY_SIZE);
  VISUAL_KBOX.REGISTERS.set_register
    (rp_reg,MEMORY_SIZE);
  VISUAL_KBOX.REGISTERS.set_register
    (hp_reg,VISUAL_KBOX.MEMORY.get_next_available ());

  VISUAL_KBOX.FLAGS.setPC(0);
  VISUAL_KBOX.FLAGS.setEQ();

  MAXPC = VISUAL_KBOX.INSTRUCTIONS.get_code_size();
}

/* ------------------------------------------------------- */

void process_data_segment (ksource_type& infile)
{
  if (data_segment_started)
  {
    cout << " ... redefinition _DATA_SEGMENT!\n";
    exit (EXIT_FAILURE);
  }
  data_segment_started = true;
  while (infile.read_line())
  {
    if (infile.is_eof())
      return;
    item = infile.get_opcode();
    if (item == "_CODE_SEGMENT")
      return;
    if (item == "_DEFINE")
    {
      string label = infile.get_operand();
      klunk value = literal2klunk(infile.get_operand());
      VISUAL_KBOX.MEMORY.define(label,value);
    }
    else if (item == ".RESERVE")
    {
      string label = infile.get_operand();
      klunk size = stoll(infile.get_operand());
      VISUAL_KBOX.MEMORY.reserve(label, size);
    }
    else
    {
      cerr << " ... inappropriate data segment item ("
           << item << ")\n";
      exit(EXIT_FAILURE);
    }
  }
}

void process_code_segment (ksource_type& infile)
{
  if (code_segment_started)
  {
    cout << " ... redefinition .CODE_SEGMENT!\n";
    exit (EXIT_FAILURE);
  }
  code_segment_started = true;
  while (infile.read_line())
  {
    if (infile.is_eof())
      return;
    item = infile.get_opcode();
    if (item == ".DATA_SEGMENT")
      return;
    if (item  == ".GLOBAL")
    {
      string label = infile.get_operand();
      VISUAL_KBOX.GLOBALS.add_global(label);
    }
    else if (item == ".EXTERN")
    {
      string label = infile.get_operand();
      VISUAL_KBOX.EXTERNS.add_extern(label);
    }
    else
    {
      process_instructions(infile);
      instructions_finished = true;
      return;
    }
  }
}

void process_instructions (ksource_type& infile)
{
  instructions_started = true;
  while (true)
  {
    if (item == "_DATA_SEGMENT")
      return;
    if (item == "_LABEL")
    {
      string label = infile.get_operand();
      label_entry le
        (label,VISUAL_KBOX.INSTRUCTIONS.get_code_size());
      VISUAL_KBOX.LABELS.add_label(le);
    }
    else if (item[0] == '_')
    {
      cerr << " ... inappropriate code segment item ("
           << item << ")\n";
      exit(EXIT_FAILURE);
    }
    else // item is an instruction opcode!
    {
      string opcode = item;
      string operand1 = infile.get_operand();
      string operand2 = infile.get_operand();
      string operand3 = infile.get_operand();
      code_entry ce(opcode,operand1,operand2,operand3);
      VISUAL_KBOX.INSTRUCTIONS.add_instruction(ce);
    }
    infile.read_line();
    if (infile.is_eof())
      return;
    else
      item = infile.get_opcode();
  }
}
```

Chapter 6

KFEDEX Implementation

KBOX Kick Boxing!



6.1 Overview

We are now coming to the final stages of our project – and hopefully some of the most enjoyable and interesting programming.

The **kfedex** implementation is the largest of the implementation files we will encounter on this journey. We must implement a total of sixty-one (61) items defined within the **kcode** instruction set.

This will be both tedious and repetitious. Because we are working with C++ objects, their members and their methods, our coding will tend to be a bit lengthy.

However, this implementation file contains only a single C++ function,

void kfedex (klunk pc, bool& halt_flag).

pc is an in-variable which identifies the location of the instruction **kfedex** is to simulate.

halt_flag is an out-variable which will indicate that a **halt** instruction has been encountered and will remain on that instruction indefinitely.

The **halt** instruction actually comes in two flavors in **visual_kbox**:

- an explicit instruction, called **halt**
- an implicit instruction,
  after executing the last instruction found in the source file

kfedex will print a notification in the console_display that a halt instruction has been encountered. It will also set the program counter to MAXPC (an implicit halt).

Note: The user must explicitly terminate the **visual_kbox** program by pressing the **exit button**.

The remaining sixty instructions must be implemented individually. Some general recommendations and comments follow in the next section

6.2 Some General Guidelines

First off, the implementation file is very long but fortunately also very repetitious. I strongly suggest that you address this phase of the project in smaller pieces that are related to one another.

I found that quite often I was able to copy the entire code from one instruction to another similar instruction with only one or two obvious modifications.

suggested instruction grouping

- nop
- i2f and f2i
- lda, ldr, and str
- mov and movi
- push and pop
- add, sub, mul, div, mod, and neg
- uadd, usub, umul, udiv, and umod
- fadd, fsub, fmul, fdiv, and fneg
- land, lor, lxor, and lnot
- band, bor, bxor, and bnot
- cmp, ucmp, and fcmp
- jmp
- jeg, jne, jgt, jlt, jge, and jle
- rol, ror, shl, shr, ashl, and ashr
- call and ret
- inc and dec
- malloc and dalloc
- get and getln
- put and putln

Second, the implementation coding for each instruction typically followed the same basic pattern of steps:

basic pattern

- verification of semantics
  reliance on **kbox** methods
- error handling (if necessary)
  error messages displayed in **console_display**
- simulation of execution
  calculations require **klunk** conversions
  comparisons also require **klunk** conversions
  and require setting **flags**
  branching requires setting **pc**

Third, a very important aspect of memory organization and memory management needs to be recognized and, at least, partially be addressed and resolved!

relationship: sp and hp

Recall that the stack grows downward from high memory and that the heap grows upward from static/global memory. Hopefully, they will never meet. But, if they do, then we have a problem.

Either the stack has grown to large (a **stack overflow**) or the heap is unable to perform a requested **malloc** and must return a NULL pointer.

Four obvious instructions require our attention: push and pop for the stack pointer and malloc and dalloc for the heap pointer.

However, pop and dalloc are instructions which can only reduce the storage but certainly never increase it. So only push and malloc remain.

- push (which adds a single **klunk** to the stack)
  will succeed if $\mathbf{sp} > \mathbf{hp}$
- malloc (with a request of **size** number of **klunks**)
  will succeed if $\mathbf{sp} > \mathbf{hp} + \mathbf{size}$

However, the comments above do not tell the entire story. While the heap may be modified by the two instructions (**malloc** and **dalloc**) and the stack may be modified by the two instructions (**push** and **pop**), they are not the only instructions. The stack pointer (**sp**) and the heap pointer(**hp**) and two integer registers like fourteen others.

Any instruction that can modify an integer register
*may be used to modify either **sp** or **hp**.*

This is a significantly larger task than focusing on just two specific instructions! As a result, I decided to limit my error checking to the two instructions I originally highlighted: **push** and **malloc**.

6.3 kfedex

kfedex.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cstring>

#include "kbox.h"
#include "kwindow.h"

using namespace std;

extern kbox_type        VISUAL_KBOX;
extern kwindow_type*    KBOX_WINDOW;
extern klunk            MAXPC;

/* ------------------------------------------------------------------ */

// kfedex.cpp

/* ------------------------------------------------------------------ */

void kwindow_type::fedex (klunk pc,bool& halt_flag)
{
  string opcode;
  string oper1,   oper2,   oper3;
  char   r_type1, r_type2, r_type3;
  klunk  r_index1,r_index2,r_index3;

  if (pc >= MAXPC)
  {
    opcode = "HALT";
    oper1 = oper2 = oper3 = "";
  }
  else
  {
    halt_flag = false;
    code_entry instruction =
      VISUAL_KBOX.INSTRUCTIONS.get_instruction(pc);
    opcode = instruction.get_opcode();
    oper1 =  instruction.get_operand1();
    oper2 =  instruction.get_operand2();
    oper3 =  instruction.get_operand3();
    VISUAL_KBOX.FLAGS.setPC(pc+1);
  }

  if (opcode == "HALT")
  {
    halt_flag = true;
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    KBOX_WINDOW->console_display->printf
      ("halt instruction!\n");
    return;
  }

  else if (opcode == "NOP")
    return;

  else if (opcode == "I2F")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid I2F operands: %s,%s\n",
        oper1.c_str(),oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register
      (oper1,flt2klunk(static_cast<flt64>(klunk2int
      (VISUAL_KBOX.REGISTERS.get_register(oper2)))));
  }
  else if (opcode == "F2I")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid F2I operands: %s,%s\n",
        oper1.c_str(),oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register
      (oper1,int2klunk(static_cast<int64>(klunk2flt
      (VISUAL_KBOX.REGISTERS.get_register(oper2)))));
  }

  else if (opcode == "LDA")
  {
    map_entry memory_info;
    if ((oper2[0] != '=') ||
        (!VISUAL_KBOX.MEMORY.is_map_entry
          (oper2.substr(1,oper2.size()-1),memory_info)))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid LDA second operand: %s\n",oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    if ((VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) &&
        (r_type1 == 'I'))
      VISUAL_KBOX.REGISTERS.set_register
        (oper1,memory_info.get_address());
    else
    {
      KBOX_WINDOW->console_display->printf
        ("invalid LDA first operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "LDR")
  {
    if ((VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) &&
        (r_type2 == 'I'))
      if (VISUAL_KBOX.REGISTERS.is_register
            (oper1,r_type1,r_index1))
      {
        VISUAL_KBOX.REGISTERS.set_register(oper1,
          VISUAL_KBOX.MEMORY.peek
            (VISUAL_KBOX.REGISTERS.get_register(oper2)));
      }
      else
      {
        printf ("invalid LDR first operand: %s\n",
          oper1.c_str());
        VISUAL_KBOX.FLAGS.setPC(MAXPC);
        return;
      }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("invalid LDR second operand: %s\n",oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "STR")
  {
    if ((VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) &&
        (r_type2 == 'I'))
      if (VISUAL_KBOX.REGISTERS.is_register
            (oper1,r_type1,r_index1))
          VISUAL_KBOX.MEMORY.poke
            (VISUAL_KBOX.REGISTERS.get_register(oper2),
             VISUAL_KBOX.REGISTERS.get_register(oper1));
      else
      {
        KBOX_WINDOW->console_display->printf
          ("invalid STR first operand: %s\n",
          oper1.c_str());
        VISUAL_KBOX.FLAGS.setPC(MAXPC);
        return;
      }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("invalid STR second operand: %s\n",
        oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }

  else if (opcode == "MOV")
  {
    if (!VISUAL_KBOX.REGISTERS.is_register
         (oper1,r_type1,r_index1))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid MOV first operand: %s\n",
        oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    if (!VISUAL_KBOX.REGISTERS.is_register
         (oper2,r_type2,r_index2))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid MOV second operand: %s\n",
        oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register
      (oper1,VISUAL_KBOX.REGISTERS.get_register(oper2));
  }
  else if (opcode == "MOVI")
  {
    if (!VISUAL_KBOX.REGISTERS.is_register
         (oper1,r_type1,r_index1))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid MOVI first operand: %s\n",
        oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    klunk imm_value = literal2klunk (oper2);
    VISUAL_KBOX.REGISTERS.set_register(oper1,imm_value);
  }

  else if (opcode == "PUSH")
  {
    if (VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1))
    {
      string sp_reg = "I12";
      klunk sp =
        VISUAL_KBOX.REGISTERS.get_register(sp_reg);
      string hp_reg = "I15";
      klunk hp =
        VISUAL_KBOX.REGISTERS.get_register(hp_reg);
      if (sp > hp)
      {
        VISUAL_KBOX.REGISTERS.set_register(sp_reg,sp-1);
        VISUAL_KBOX.MEMORY.poke
          (sp-1,VISUAL_KBOX.REGISTERS.get_register(oper1));
      }
      else
      {
        KBOX_WINDOW->console_display->printf
          ("stack overflow\n");
        VISUAL_KBOX.FLAGS.setPC(MAXPC);
        return;
      }
    }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("invalid PUSH operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "POP")
  {
    if (VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1))
    {
      string sp_reg = "I12";
      klunk sp =
        VISUAL_KBOX.REGISTERS.get_register(sp_reg);
      VISUAL_KBOX.REGISTERS.set_register(oper1,
        VISUAL_KBOX.MEMORY.peek(sp));
      VISUAL_KBOX.REGISTERS.set_register(sp_reg,sp+1);
    }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("invalid POP operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }

  else if (opcode == "ADD")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid ADD operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2)) +
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "SUB")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid SUB operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2)) -
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "MUL")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid MUL operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2)) *
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "DIV")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid DIV operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    int64 divisor = klunk2int
      (VISUAL_KBOX.REGISTERS.get_register(oper3));
    if (divisor == 0)
    {
      KBOX_WINDOW->console_display->printf
        ("division by 0 not permitted!\n");
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2)) /
      divisor));
  }
  else if (opcode == "MOD")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
            (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
            (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
            (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid MOD operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    int64 divisor = klunk2int
      (VISUAL_KBOX.REGISTERS.get_register(oper3));
    if (divisor == 0)
    {
      KBOX_WINDOW->console_display->printf
        ("division by 0 not permitted!\n");
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2)) %
      divisor));
  }
  else if (opcode == "NEG")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
            (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid NEG operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,int2klunk(
      - klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper1))));
  }

  else if (opcode == "UADD")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
            (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
            (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid UADD operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) +
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "USUB")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid USUB operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) -
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "UMUL")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid UMUL operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) -
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "UDIV")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid UDIV operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    klunk divisor =
      VISUAL_KBOX.REGISTERS.get_register(oper3);
    if (divisor == 0)
    {
      KBOX_WINDOW->console_display->printf
        ("division by 0 not permitted!\n");
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) /
      divisor);
  }
  else if (opcode == "UMOD")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid UMOD operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    klunk divisor =
      VISUAL_KBOX.REGISTERS.get_register(oper3);
    if (divisor == 0)
    {
      KBOX_WINDOW->console_display->printf
        ("division by 0 not permitted!\n");
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) /
      divisor);
  }

  else if (opcode == "FADD")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FADD operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,flt2klunk(
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper2)) +
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "FSUB")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FSUB operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1, flt2klunk(
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper2)) -
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "FMUL")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1, r_type1, r_index1)) ||
         (r_type1 != 'F') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2, r_type2, r_index2)) ||
         (r_type2 != 'F') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3, r_type3, r_index3)) ||
         (r_type3 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FMUL operands: %s, %s, %s\n",
          oper1.c_str(), oper2.c_str(), oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1, flt2klunk(
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper2)) *
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper3))));
  }
  else if (opcode == "FDIV")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1, r_type1, r_index1)) ||
         (r_type1 != 'F') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2, r_type2, r_index2)) ||
         (r_type2 != 'F') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3, r_type3, r_index3)) ||
         (r_type3 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FDIV operands: %s, %s, %s\n",
          oper1.c_str(), oper2.c_str(), oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    flt64 divisor =
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper3));
    if (divisor == 0.0)
    {
      KBOX_WINDOW->console_display->printf
        ("division by 0 not permitted!\n");
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,flt2klunk(
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper2)) /
      divisor));
  }
  else if (opcode == "FNEG")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FNEG operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,flt2klunk(
      - klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper1))));
  }

  else if (opcode == "LAND")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid LAND operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) &&
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "LOR")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper3,r_type3,r_index3)) ||
        (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
         ("invalid LOR operands: %s, %s, %s\n",
           oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) ||
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "LXOR")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1,r_type1,r_index1)) ||
         (r_type1 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2,r_type2,r_index2)) ||
         (r_type2 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3,r_type3,r_index3)) ||
         (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
         ("invalid LXOR operands: %s, %s, %s\n",
           oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      (VISUAL_KBOX.REGISTERS.get_register(oper2) &&
      (!VISUAL_KBOX.REGISTERS.get_register(oper3))) ||
      (VISUAL_KBOX.REGISTERS.get_register(oper3) &&
      (!VISUAL_KBOX.REGISTERS.get_register(oper2))));
  }
  else if (opcode == "LNOT")
  {
   if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
         ("invalid LNOT operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      !VISUAL_KBOX.REGISTERS.get_register(oper1));
  }

  else if (opcode == "BAND")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1,r_type1,r_index1)) ||
         (r_type1 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2,r_type2,r_index2)) ||
         (r_type2 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3,r_type3,r_index3)) ||
         (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
         ("invalid BAND operands: %s, %s, %s\n",
           oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) &
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "BOR")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1,r_type1,r_index1)) ||
         (r_type1 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2,r_type2,r_index2)) ||
         (r_type2 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3,r_type3,r_index3)) ||
         (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
         ("invalid BOR operands: %s, %s, %s\n",
           oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) |
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "BXOR")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
           (oper1,r_type1,r_index1)) ||
         (r_type1 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper2,r_type2,r_index2)) ||
         (r_type2 != 'I') ||
         (!VISUAL_KBOX.REGISTERS.is_register
           (oper3,r_type3,r_index3)) ||
         (r_type3 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid BXOR operands: %s, %s, %s\n",
          oper1.c_str(),oper2.c_str(),oper3.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      VISUAL_KBOX.REGISTERS.get_register(oper2) ^
      VISUAL_KBOX.REGISTERS.get_register(oper3));
  }
  else if (opcode == "BNOT")
  {
   if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid BNOT operand: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    VISUAL_KBOX.REGISTERS.set_register(oper1,
      ~VISUAL_KBOX.REGISTERS.get_register(oper1));
  }

  else if (opcode == "CMP")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid CMP operands: %s, %s\n",
          oper1.c_str(),oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    int64 diff =
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper1)) -
      klunk2int(VISUAL_KBOX.REGISTERS.get_register(oper2));
    if (diff == 0LL)
      VISUAL_KBOX.FLAGS.setEQ ();
    else if (diff > 0LL)
      VISUAL_KBOX.FLAGS.setGT ();
    else // (diff < 0LL)
      VISUAL_KBOX.FLAGS.setLT ();
  }
  else if (opcode == "UCMP")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'I') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'I'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid UCMP operands: %s, %s\n",
          oper1.c_str(),oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    int64 diff =
      VISUAL_KBOX.REGISTERS.get_register(oper1) -
      VISUAL_KBOX.REGISTERS.get_register(oper2);
    if (diff == 0LL)
      VISUAL_KBOX.FLAGS.setEQ ();
    else if (diff > 0LL)
      VISUAL_KBOX.FLAGS.setGT();
    else // (diff < 0LL)
      VISUAL_KBOX.FLAGS.setLT();
  }
else if (opcode == "FCMP")
  {
    if ((!VISUAL_KBOX.REGISTERS.is_register
          (oper1,r_type1,r_index1)) ||
        (r_type1 != 'F') ||
        (!VISUAL_KBOX.REGISTERS.is_register
          (oper2,r_type2,r_index2)) ||
        (r_type2 != 'F'))
    {
      KBOX_WINDOW->console_display->printf
        ("invalid FCMP operands: %s, %s\n",
          oper1.c_str(),oper2.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
    flt64 diff =
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper1)) -
      klunk2flt(VISUAL_KBOX.REGISTERS.get_register(oper2));
    if (diff == 0.0)
      VISUAL_KBOX.FLAGS.setEQ ();
    else if (diff > 0.0)
      VISUAL_KBOX.FLAGS.setGT ();
    else // (diff < 0.0)
      VISUAL_KBOX.FLAGS.setLT ();
  }

  else if (opcode == "JMP")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      VISUAL_KBOX.FLAGS.setPC(
        VISUAL_KBOX.LABELS.get_location (oper1));
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }

  else if (opcode == "JEQ")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (VISUAL_KBOX.FLAGS.getEQ())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      { }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "JNE")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (!VISUAL_KBOX.FLAGS.getEQ())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      { }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "JGT")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (VISUAL_KBOX.FLAGS.getGT())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      { }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "JLT")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (VISUAL_KBOX.FLAGS.getLT())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      {  }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "JGE")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (!VISUAL_KBOX.FLAGS.getLT())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      {  }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
  else if (opcode == "JLE")
  {
    if (VISUAL_KBOX.LABELS.is_label(oper1))
      if (!VISUAL_KBOX.FLAGS.getGT())
        VISUAL_KBOX.FLAGS.setPC(
             VISUAL_KBOX.LABELS.get_location(oper1));
      else
      {  }
    else
    {
      KBOX_WINDOW->console_display->printf
        ("undefined label: %s\n",oper1.c_str());
      VISUAL_KBOX.FLAGS.setPC(MAXPC);
      return;
    }
  }
```

```
else if (opcode == "ROL")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid ROL register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  klunk bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  klunk parta = bit_pattern << shift;
  klunk partb = bit_pattern >> (64-shift);
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,parta | partb);
}
else if (opcode == "ROR")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid ROR register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  klunk bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  klunk parta = bit_pattern >> shift;
  klunk partb = bit_pattern << (64-shift);
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,parta | partb);
}
else if (opcode == "SHL")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid SHL register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  klunk bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,bit_pattern << shift);
}
else if (opcode == "SHR")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid SHR register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  klunk bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,bit_pattern >> shift);
}
else if (opcode == "ASHL")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid ASHL register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  klunk bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,bit_pattern << shift);
}
else if (opcode == "ASHR")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid ASHR register: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  // note: bit pattern must be considered as
  // a signed and not an unsigned value
  int64 bit_pattern =
    VISUAL_KBOX.REGISTERS.get_register(oper1);
  string imm = oper2;
  int shift = stoi(imm) & 0X3F;
  VISUAL_KBOX.REGISTERS.set_register
    (oper1,bit_pattern >> shift);
}

else if (opcode =="CALL")
{
  string rp_reg = "I14";
  if (VISUAL_KBOX.LABELS.is_label(oper1))
  {
    VISUAL_KBOX.REGISTERS.set_register
      (rp_reg,VISUAL_KBOX.FLAGS.getPC());
    VISUAL_KBOX.FLAGS.setPC
      (VISUAL_KBOX.LABELS.get_location(oper1));
  }
  else
  {
    KBOX_WINDOW->console_display->printf
      ("undefined label: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
}
else if (opcode == "RET")
{
  string rp_reg = "I14";
  VISUAL_KBOX.FLAGS.setPC
    (VISUAL_KBOX.REGISTERS.get_register(rp_reg));
}

else if (opcode == "INC")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid INC operand: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  VISUAL_KBOX.REGISTERS.set_register(oper1,
    VISUAL_KBOX.REGISTERS.get_register(oper1)+1);
}
else if (opcode == "DEC")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid DEC operand: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  VISUAL_KBOX.REGISTERS.set_register(oper1,
    VISUAL_KBOX.REGISTERS.get_register(oper1)-1);
}

else if (opcode == "MALLOC")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I') ||
      (!VISUAL_KBOX.REGISTERS.is_register
        (oper2,r_type2,r_index2)) ||
      (r_type2 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid MALLOC operands: %s, %s\n",
      oper1.c_str(),oper2.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  string sp_reg = "I12";
  klunk sp =
    VISUAL_KBOX.REGISTERS.get_register(sp_reg);
  string hp_reg = "I15";
  klunk hp =
    VISUAL_KBOX.REGISTERS.get_register(hp_reg);
  klunk no_klunks =
    VISUAL_KBOX.REGISTERS.get_register(oper2);
  if ((sp - hp) > no_klunks)
  {
    VISUAL_KBOX.REGISTERS.set_register(oper1,hp);
    VISUAL_KBOX.REGISTERS.set_register
      (hp_reg,hp+no_klunks);
  }
  else
    VISUAL_KBOX.REGISTERS.set_register(oper1,0);
}
else if (opcode == "DALLOC")
{
  if ((!VISUAL_KBOX.REGISTERS.is_register
        (oper1,r_type1,r_index1)) ||
      (r_type1 != 'I') ||
      (!VISUAL_KBOX.REGISTERS.is_register
        (oper2,r_type2,r_index2)) ||
      (r_type2 != 'I'))
  {
    KBOX_WINDOW->console_display->printf
      ("invalid DALLOC operands: %s, %s\n",
      oper1.c_str(),oper2.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  else
  {
     // yes, do nothing!
     // this is minimal memory management
     // more like NON-EXISTENT memory management
  }
}

else if (opcode == "GET")
{
  string sp_reg = "I12";
  VISUAL_KBOX.REGISTERS.set_register(sp_reg,
    VISUAL_KBOX.REGISTERS.get_register(sp_reg)-1);
  string fmt = oper1;
  str64 buffer = (char*) malloc (LITERAL_SIZE);
  if (fmt == "INT")
  {
    popup("enter integer: ",buffer);
    // immediately echo the input buffer
    // to the console display
    KBOX_WINDOW->console_display->printf
      ("%s\n",buffer);
    int64 data = atoll(buffer);
    VISUAL_KBOX.MEMORY.poke
      (VISUAL_KBOX.REGISTERS.get_register(sp_reg),
       int2klunk(data));
  }
  else if (fmt == "FLT")
  {
    popup("enter real: ",buffer);
    KBOX_WINDOW->console_display->printf
      ("%s\n",buffer);
    flt64 data = atof(buffer);
    VISUAL_KBOX.MEMORY.poke
      (VISUAL_KBOX.REGISTERS.get_register(sp_reg),
       flt2klunk(data));
  }
  else if (fmt == "CHR")
  {
    popup("enter single character: ",buffer);
    KBOX_WINDOW->console_display->printf
      ("%s\n",buffer);
    chr64 data = buffer[0];
    VISUAL_KBOX.MEMORY.poke
      (VISUAL_KBOX.REGISTERS.get_register(sp_reg),
       chr2klunk(data));
  }
  else if (fmt == "STR")
  {
    popup("enter string: ",buffer);
    KBOX_WINDOW->console_display->printf
      ("%s/n",buffer);
    VISUAL_KBOX.MEMORY.poke
      (VISUAL_KBOX.REGISTERS.get_register(sp_reg),
       str2klunk(buffer));
  }
  else
  {
    KBOX_WINDOW->console_display->printf
      ("invalid format entry: %s\n",fmt.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
}
else if (opcode == "GETLN")
{
  // popup window in graphical user interface
  // makes this instruction essentially obsolete!
  KBOX_WINDOW->console_display->printf
    ("\n");
}

else if (opcode == "PUT")
{
  string sp_reg = "I12";
  klunk data =
    VISUAL_KBOX.MEMORY.peek
      (VISUAL_KBOX.REGISTERS.get_register(sp_reg));
  string fmt = oper1;
  if (fmt == "INT")
    KBOX_WINDOW->console_display->printf
      ("%lld",klunk2int(data));
  else if (fmt == "FLT")
    KBOX_WINDOW->console_display->printf
      ("%f",klunk2flt(data));
  else if (fmt == "CHR")
    KBOX_WINDOW->console_display->printf
      ("%c",klunk2chr(data));
  else if (fmt == "STR")
    KBOX_WINDOW->console_display->printf
      ("%s",klunk2str(data));
  else if ((fmt == "HEX") ||
           (fmt == "PTR"))
    KBOX_WINDOW->console_display->printf
      ("%016llx",data);
  else
  {
    KBOX_WINDOW->console_display->printf
      ("invalid format entry: %s\n",oper1.c_str());
    VISUAL_KBOX.FLAGS.setPC(MAXPC);
    return;
  }
  VISUAL_KBOX.REGISTERS.set_register(sp_reg,
    VISUAL_KBOX.REGISTERS.get_register(sp_reg)+1);
}
else if (opcode == "PUTLN")
  KBOX_WINDOW->console_display->printf
    ("\n");

else
{
  KBOX_WINDOW->console_display->printf
    ("unrecognized instruction: %s\n",opcode.c_str());
  VISUAL_KBOX.FLAGS.setPC(MAXPC);
  return;
}
}
```

Chapter 7

VISUAL KBOX

The KBOX Multi-Lingual Learning Program!

Have you ever faced language barriers?
Are you unable to find any e-learning courses in your native language?
Do you struggle when finding non-english MOOCs?
Your search ends here!
KBOX
A Multi-lingual Collaborative E-Learning Platform
γεια
HALLO
XIN CHÀO
नमस्ते
MERHABA
SVEIKI
привет
привіт
HOLA
你好
こんにちは
مرحبا

7.1 Overview

After all our previous work, in the original text **Fun With Programming Language** and in this text **Visual_KBOX**, this chapter is essentially an afterthought and minimal in any additional content.

The heavy lifting was done in the chapter focusing on **kwindow_type**. Graphical user interface programming is detail intensive:

- first in the design of the user interface
  layout of the various widgets
  define content and attributes for each widget
- second in the implementation of the wait-for and call-back mechanisms
  remember static call-back activates
  widget method to perform actual call-back algorithm

As a result this entire chapter can be summarized in the following important sequence of steps to be included in a main driver program:

- **source file name** is specified in command line (argv[1])
- call **ksetup** (source_file_name)
- create KBOX_WINDOW and initialize its contents
- show the KBOX_WINDOW!
- turn control over to FLTK software (Fl::run())

7.2 visual_kbox

virtual_kbox.cpp

```
#include <cstdlib>
#include <iostream>
#include <FL/Fl.H>

#include "kbox.h"
#include "kwindow.h"
#include "ksource.h"
#include "ksetup.h"

using namespace std;

/* ------------------------------------------------------------------ */

kbox_type        VISUAL_KBOX;
ksource_type     KBOX_SOURCE;
kwindow_type*    KBOX_WINDOW;

/* ------------------------------------------------------------------ */

int main (int argc, char* argv[])
{
  string source_file_name;
  if (argc >= 2)
    source_file_name = argv[1];
  else
  {
    cerr << "    ... kcode source file required!\n";
    exit(EXIT_FAILURE);
  }

  setup_kbox(source_file_name);

  KBOX_WINDOW = new kwindow_type { };
  KBOX_WINDOW->show();

  return Fl::run();
}
```

7.3 Concluding Remarks

This project has been especially rewarding for me personally. I wanted to experience programming with a graphical toolkit and FLTK provided me with that opportunity.

It was much different than I had originally expected. It was more research and reading a manual than I had anticipated; and it was significantly less algorithmic than I had anticipated. Programming skills seem to be more concentrated in the call-back mechanisms.

And the user-oriented focus of graphical user interfaces creates a whole different approach to programming than more traditional programming. I found that very eye opening!

It was a very similar "aha" moment to when I wrote my first recursive descent parser – the main program essentially just called the start procedure and then immediately checked if it had successfully terminated!

With **visual_kbox** it was layout a bunch of widgets in a window and define a bunch of call-back methods. The main driver basically just starts the application and the user simply hangs on for the ride!

But do not get me wrong here! In the end it was a ***very enjoyable*** ride!

It felt different than more traditional programming – not having to prompt the user for input when and where it seemed appropriate in the code. Rather it seemed more reactive – What do I do if the user requests *this*? Or what do I do if the user request *that*?

And feeling different does not mean one is better or worse! Both strike me as challenging in different ways. And both give the programmer the same feeling of accomplishment at the end of a project..