Aarti Paan, Dhruti Pandya, Matthew Posa, and Chuhui Qiu

Professor Morris Lancaster

CS6461 Section 10 Spring 2022

13 March 2022

**Group Project Phase 1 Design Note**

This design note documents the structure of the simulator project, the classes, methods, functions, and data structures implemented, as well as any external dependencies used for this project.

# General Structure

The project consists of eight source code modules separated into eight python scripts, one compiled executable file, one User Guide document, and one Design Note document.

***Executable File.* CS 6461 Project Phase 2 Executable.exe** is the main file built for an x86\_64 Windows machine. It is built using the command “pyinstaller --onefile CSCI6461\_Project\_main.py” and includes all eight python scripts as well as any external dependencies.

***Main.* CSCI6461\_Project\_main.py** is the script that serves as the entry point for running the program, it can either be imported as a library or run directly to start the program.

1. Imports: the script imports module CSCI6461\_Project\_gui\_interface and CSCI6461\_Project\_data from the other files, no external dependencies are imported.
2. The script defines one function “main()”, it takes no arguments and does not return; its function is to initialize the memory data, open the GUI interface, and enter a loop, in which it will accept any user input into the Python terminal and attempt to execute it with exec().
3. Global variables from globals() and “data” from CSCI6461\_Project\_data are used as locals for exec(); Exceptions are caught and merely printed then ignored.
4. Safety: The Python terminal is used as an engineer’s field console, and as a result no security measure is in place. It is possible to execute “rm -rf” there, so users should know what they are doing before attempting dangerous operations.

***GUI Classes.* CSCI6461\_Project\_classes.py** is the script containing definitions for the primary GUI elements: bit, bitString, and labeledBitString, and their related interface methods. It can be imported as a GUI library.

1. Imports: the script imports the external module “tkinter” and its submodule “ttk”, both are built-in to Python 3.7.
2. Bit Class: the fundamental class for a single button that acts as a single-bit register, it holds a single number 0 or 1 to show its content.
3. Method \_\_init\_\_(self, value=0) initializes an instance of a Bit by setting three properties.

i. “btn” for holding a ttk.Button instance

ii. “trigs” for holding a dictionary of functions to be triggered

iii. “v” for holding a binary value, optionally specified by the “value” parameter.

1. Method create(self, frame, x=0, y=0, \*\*kwargs) creates the actual button onto a Tkinter frame, it would optionally take positional arguments x and y, and any additional keyword-ed arguments that are accepted by ttk.Button(), the Bit instance must have been initialized first.
2. Method destroy(self) destroys said button, allowing it to be recreated.
3. Methods value(self) and value\_get(self) returns the binary bit stored within the instances as an integer.
4. Method value\_set(self, value=0, trigger=True) may be called to set the binary value stored in the instance to 0 or 1; an optional boolean “trigger” parameter is used to specify whether this update will cause the “trigs” dictionary of functions to be called.
5. Method flip(self) toggles the value stored in the instance between 0 and 1.
6. Method update(self, trigger=True) reconfigures the text on the button to reflect its current value; if the boolean “trigger” parameter is true, each function within the “trigs” dictionary will be called.
7. BitString Class: a continuous “string” of Bit instances to represent a multi-bit register 
8. Method \_\_init\_\_(self, count=1) works similar to that for the Bit class, but instead of setting an initial value, all Bit instances are assumed to be 0, and its length is set by the parameter “count”.
9. Method create(self, frame, x=0, y=0, toggleAble=True, \*\*kwargs) create(self, frame, text="", x=0, y=0, gap=0, padx=0, pady=0, toggleAble=True, numLabels=True, \*\*kwargs) works similar to that for the Bit class, with an additional argument “toggleAble” for configuring whether the instances can be clicked and toggled.
10. Method destroy(self), update(self,trigger=True), value(self), value\_get(self), value\_set(self, value, trigger=True) works the same way as Bit, except value\_set() now configures the string of bits to be the Big-Endian, binary representation of the given integer “value”, if the value cannot be fully represented due to overflowing or negative input, the nearest binary representation will be taken and the remainder will be returned as an integer.
11. bitStyle(self, \*\*kwargs) is a shorthand way to quickly configure all buttons within the Bit instance to the same ttk style. This function is currently unused.
12. LabeledBitString class: a class inheriting the BitString class, it additionally holds up to three ttk label widgets for displaying the name and value of the register it represents.
13. Method \_\_init\_\_(self, value=0) initializes the instance similar to BitString, and also initializes three properties holding ttk label widgets.

i. “labelTxt” for holding the name of the instance.

ii. “labelHex” for holding a hexadecimal representation of the value of the instance.

iii. “labelTen” for holding a decimal representation of the value of the instance.

1. Method create(self, frame, text="", x=0, y=0, gap=0, padx=0, pady=0, toggleAble=True, numLabels=True, \*\*kwargs) places the instance onto a Tkinter window; besides the normal parameters explained above, it also includes “text” to specify the name of the instance, integer “gap” to place before the buttons for widget alignment, and whether the numeric representation labels should be generated.
2. Method text\_set(self, text="") sets labelTxt to the newly given string of text.
3. Method create(frame=frame, x=x+1+gap, y=y, width=2, toggleAble=toggleAble) creates the bitString, using the parameters to ensure that no elements overlap.
4. Method destroyLabel(self, label) removes a particular given label from the Tkinter window, this is mainly used as a helper function for destroy(self).
5. Method destroy(self) and update(self, trigger=True) works similarly to that of the other two classes. One addition is that update() now also updates the numeric representation labels of the instance to reflect the new number whenever they change.

***Interface GUI.* CSCI6461\_Project\_gui\_interface.py** is the script containing definitions for creating the main GUI window of the program. It can be imported as a GUI library.A picture containing text

Description automatically generated

1. Imports: the script imports the external module “sys”, “tkinter” and its submodule “ttk”, as well as “filedialog” which are all built-in to Python 3.7. it also imports some other libraries written and defined further in the program such as “CSCI6461\_Project\_classes”, “CSCI6461\_Project\_gui\_memory”, “CSCI6461\_Project\_execution”, “CSCI6461\_Project\_data”, and “CSCI6461\_Project\_gui\_io”.
2. Modules
   1. The function def guiInterface() creates a main GUI window of our project. The data['windowInterface'] is used to create the main GUI window of our program named “windowInterface” & titled as “CSCI6461 Project Machine Interface”. When the window is closed, sys.exit() is called to exit the program.
   2. This main window is further split into two further frames: “registersGPRFrame” on the left and “controlsFrame” on the right.
      1. Frames are created using the sub module ttk and placed using module grid.
      2. Positions are defined as per x(column) and y(row).
      3. x=0,y=0 is the top left, going right/down is the positive x/y direction
   3. Further data['registersGPRFrame'] creates a “registersGPRFrame” containing general purpose registers and index registers. There are four 16-bit GPRs and three 16-bit IXRs.
      1. Four GPRs are located from (X=0) & (Y=0) to (X=0) & (Y=3). Using a for loop all the GPRs of 16 bits are created.
      2. Three IXRs are located from (X=0) & (Y=5) to (X=0) & (Y=7). Using a for loop all the IXRs of 16 bits are created.
   4. A controlsFrame is created on the right of the registersGPRFrame for creating remaining registers. It is divided into three parts (a) “registersMainFrame” location column=0 & row=0, containing PC, MAR, MBR and IR. (b) “conditionsFrame” location column=0 & row=2 containing MFR, CC, and HLT registers. (c) Lastly, optionsFrame whose location is column=0 & row=4, which contains step, Run, Store, ST+, Load, Initialize and Memory buttons.
      1. PC, MAR, MBR and IR registers are created using the instance of labeledBitString (defined in CSCI6461\_Project\_classes) created to represent the registers.. And are placed in the positions ranging from (X=0) & (Y=0) to (X=0) & (Y=3) respectively of the registersMainFrame. Space is added between registers using spacers at row=1 and row=3 where column=0.
      2. PC and MAR are 12 bit registers while MBR and IR are 16 bits registers. Hence, a gap of 4 is added between text and bits for PC and MAR.
      3. Creating more instances of labeledBitString, MFR(X=0 & Y=0), CC(X=8 & Y=0), and HLT(X=14 & Y=0) registers are created. Space is added in between the registers using spacers at column=7 and column=13 where row=0. MFR, CC and HALT are 4, 2 and 1 bit registers respectively.
      4. In optionFrame buttons such as step, Run, Store, ST+, Load, I/O, Initialize and Memory button are created with the instances of the button from the ttk module.
3. Methods:
   1. store(plus=False): This is the function which is called when a user clicks on store or st+ buttons. It takes the address from the MAR register and value from the MBR register and writes it down to the memory location for the store button. If it is a st+ button then, it sets the value of MAR to address+1.
   2. ProgramLoad(): This function is used to load a text file to initialize values into memory. It will set GPR, IXR, PC,MAR,MBR,IR,MFR,CC,HALT as Zero (0). And read the uploaded file provided by the user. If the file is not readable or if it is not appropriate then it throws an exception otherwise it loads the provided hexadecimal value into memory. And call memoryBlockUpdate function to reflect the changes on the GUI.

***Memory GUI.*****CSCI6461\_Project\_gui\_memory.py** is the script containing definitions relating to the memory viewer/editor GUI. It can be imported as a GUI library and it is primarily for Debugging and visualizing changes, so using this GUI is not absolutely necessary.

Table

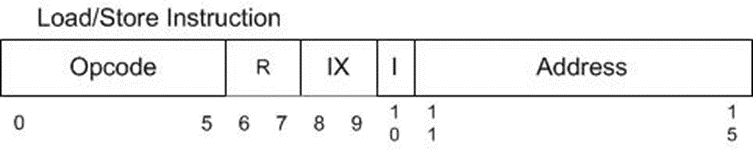
Description automatically generated

1. Imports: The module imports the external module “tkinter” and its submodule “ttk” which are built-in to Python 3.7. It also imports “CSCI6461\_Project\_classes” and “CSCI6461\_Project\_data”.
2. Modules:
3. “memoryPage” frame at the top of the window is a single instance of labeledBitString “memoryBlockNum” with the first one and last four buttons disabled, it functions as a page navigator showing the page number and the address of the currently shown memory locations. When the number in the navigator is changed by pressing the buttons, the frame below will be updated to show the words in the destination memory page.
4. “memoryCont” frame in the remaining parts of the window holds 16 labeledBitString instances “memoryEntry”, each of them holds the value to one 16-bit word in memory and displays its value and its word address.
5. Methods:
6. windowMemory\_onClose() : Whenever the memory editor is closed this function gets triggered. It destroys and removes the record of the window and re-enables the button on the main interface for reopening the editor.
7. MemoryWriteTrig(address) : This function is used to write values into a memory word from the memory editor. If the memory editor is not opened, it will not perform any operation.
8. MemoryBlockUpdate(): This function is used to refresh the page when the memory editor’s page number changes. It calculates the new page number from “memoryBlockNum” register, and iterates through each of the “memoryEntry” instances while setting their newly pointed-to address, new trigger functions to write to memory when their value changes, and change their display value to that currently stored in the memory.
9. guiMemoryJump(blockNum) : This function either calls guiMemory() if data lacks “windowMemory” or waits a second for it to change and reads it again and sets “memoryBlockNum” to the provided argument.
10. guiMemory() : This function creates the memory viewer/editor GUI window. The open button on the main interface will also be disabled to prevent opening multiple memory windows.
11. Creates a separate window to view/edit the memory named "CSCI6461 Project Machine Memory". data[windowMemory] is the main GUI window for memory. It splits into two further frames “memoryBlockFrame” & “memoryContFrame”.
12. if the window is closed, destroy it by calling windowMemory\_onClose() function.
13. Configures the trigger functions dictionary within “memoryBlockNum” to be able to update the memory content display when needed.

***Global Data.* CSCI6461\_Project\_data.py** is the script containing definition and initialization for a single variable named “data” to be used by the console and other modules, it will contain all information held by the program. This includes all of memory, all registers, and all gui elements. This module can be imported by itself, but without utilization from other modules, it is practically useless.

***Execution.* CSCI6461\_Project\_execution.py** is the script containing definitions for all instruction-related operations in the simulator program. Currently it only contains functions to execute HLT/LDR/LDA/LDX/STR/STX instructions, to throw machine faults, the function to be called when the “step” or “run” button is pressed on the main user interface, and helper functions to assist them. It is possible but not useful to import this module by itself.

1. Imports: It imports “CSCI6461\_Project\_gui\_memory” , “CSCI6461\_Project\_data”, “CSCI6461\_Project\_classes”, CSCI6461\_Project\_gui\_cache” and “CSCI6461\_Project\_gui\_io”.
2. Methods:
   1. fault(id) : This function handles machine faults. It first sets the MFR in the main interface to the corresponding fault id, stores the erroneous PC to memory location 4, then loads the address at memory location 1 into PC so the machine can continue running.
   2. HLT() : This function stops the machine by setting the HALT register to 1.
   3. splitInstructionLoadStore(instruction) :Instruction is a 16-bit integer (0-65535).This function is used to split the 16-bit instruction into opcode(6), r(2), ix(2), i(1), address(5) in this order for load and store instructions.



* 1. splitInstuctionArithmetic(instruction) : This function splits the opcode for use in ArithmeticInstExec(instruction), which is explained a few points below.
  2. splitInstructionShift(instruction) : This function does similar to the previous function but is used in ShiftInstExec(instruction).
  3. LoadStoreInstExec(instruction) : This function is used to execute load/store instructions (LDR/LDA/LDX/STR/STX). Initialize effective address (EA) and return value: if index register value is greater than zero then it adds address and IXR value to calculate EA. Furthermore, if it is LD instructions (LDR/LDA/LDX) then get Value From Memory by calling readFromMemory(EA, False) function into GPR and IXR. If opcode represents store instruction then it gets the value from GPR and IXR and writes the data into memory by calling writeToMemory(EA, value, indirect=i) function. This function returns read value if instruction is successfully loaded, returns 0 if instruction is successfully written and returns -(fault ID)-1 if any fault occurs.
  4. AddSubInstExec(instruction) : This function performs addition and/or subtraction instructions on the opcode, then sets GPR and CC to the proper numbers, then returns 0 to prevent errors.
  5. TransferInstExec(instruction) : This function may set PC to EA depending on if
  6. execute(instruction) : This function executes an instruction which is a 16-bit integer (0-65535). It returns true if executed normally and returns False if halted or faulted. Internally it checks the opcode if it is in [1,2,3,33,34] then calls LoadStoreInstExec(instruction) and returns true if it is 0 then it returns false.
  7. singleStep() : Get the value of PC and read the instruction using readFromMemory(address) function and check if instruction is executable then update IR and call execute(instruction) function. If the execute(instruction) function is successfully executed without any fault then it increases the PC by one.
  8. multiStep() : This function keeps calling singleStep until it stopped (i.e. HALT = 1)

***Cache.* CSCI6461\_Project\_gui\_cache.py** is the script that handles and creates the cache window, finds the line the user input and prepares the information to be written in memory.

1. Imports: The module imports the external module “tkinter” and its submodule “ttk”, which are native to Python 3.7. It also imports “CSCI6461\_Project\_data” and “CSCI6461\_Project\_classes”.

# cache structure:  
# data['cache']  
# ├data['cache'][lineNum] with lineNum in [0,15] for each cache line  
# │├data['cache'][lineNum][wordNum] with wordNum in [0,15] for the 16-bit content of each word  
# │├data['cache'][lineNum][16] is a 8-bit tag of the cache line  
# │├data['cache'][lineNum][17] is the initialization bit of the cache line  
# │└data['cache'][lineNum][18] is the dirty bit of the cache line  
# └data['cache'][16] is an integer pointing to the cache line number to overwrite next

1. Methods:
   1. windowCache\_onClose() : This function checks to see if the cache window is open and greys out the cache button if it is.
   2. cacheWriteTrig(lineNum, wordNum) : This function looks a pointer and writes to the respective part of the cache. This is either one of the 16 words or tag, init bit or dirty bit.
   3. cacheLineUpdate() : This function sets lineNum to be used in other functions, updates triggers related to the cache window and calls another function [guiMemoryJump(blockNum)] when the Write to Memory button is pressed.
   4. cache\_next\_update() : This function triggers when line number changes and sets it to the new line.
   5. guiCache() : constructs each GUI elements of the cache window.

# GUI Structure:  
# windowCache  
# ├cacheLineFrame  
# │├cacheLineNum  
# │├cacheNextNum  
# │└autoShowCacheBtn  
# ├cacheUtilFrame  
# │├guiMemoryJumpBtn  
# │└cleanCacheLineBtn  
# ├cachePropertyFrame  
# │├cacheInit  
# │├cacheDirty  
# │└cacheTag  
# └cacheContFrame  
# └cacheEntry[0,15]

* 1. cleanCacheLine(cacheLine=-1, update=True) : This function cleans the cache and is the only time the machine writes to memory. It takes the current line, cleans it if needed and calls cacheLineUpdate() to update the line.
  2. checkHit(tag) : This function checks if the line the machine writes to is in cache and returns the line. Otherwise returns -1, which is the safe int.
  3. loadCache(tag, update=True) : This function loads tag, puts it in line 16 and returns cacheLine.
  4. cacheMemoryRead(address) : This function reads the value from the line given from the argument. It first tries to read it from cache, then reads from memory if cache didn’t have it.
  5. cacheMemoryWrite(address, value): This function writes the previously read value to the cache.
  6. readFromMemory(address, indirect=False) : This function is used to get the value from memory and update/write the registers which are MAR and MBR accordingly and return the result.
  7. writeToMemory(address, value, indirect=False, checkReserve=True) : This function writes the value to address in memory, while updating MAR/MBR accordingly. It gets the effective memory address into MAR and tries to write the value to MBR and effective address in memory and calls memoryBlockUpdate() to refresh the page. This function returns -(fault ID)-1 if fault occurs otherwise return 0.

***I/O:-* CSCI6461\_Project\_gui\_io.py** is the script containing definitions for Input/Output related operations in the simulator program. I/O operations communicate with the peripherals attached to the computer system.

1. Imports: the script imports the external module “tkinter” and its submodule “ttk”, also it imports “CSCI6461\_Project\_data”.
2. Methods:
   1. windowIo\_onClose() :This will trigger function when I/O interface is closed. This function destroy and remove record of the window and re-enable the button in optionsFrame for reopening i/o.
   2. ioUpdate() : This function will get trigger when saved text changes, need to update the widgets and do nothing if io window isn't open. This function used to get the printer and keyboard widget, enable & disable editing and scroll to the very end to show the text.
   3. guiIo(): This function creates interface for input/output GUI. This main window is further split into following frames:
      * 1. ioFrame
        2. ioPrinterLabel
        3. ioPrinterText
        4. ioKeyboardLabel
        5. ioKeyboardText
        6. ioKeyboardBtn

The data['windowIo'] is used to create a separate window for the I/O & titled as “CSCI6461 Project Input/Output”. This function is used to create a frame to hold all i/o widgets, create the tkinter label and textbox for the console printer and keyboard, get style settings for windowIo and if window is closed, destroy it.

* 1. splitInstuctionIo(instruction): This function is used to split 16-bit instructions into

opcode(6), r(2), unused(3), devid(5) in that order for I/O instructions. The unused(3) bits of the instruction are ignored

A screenshot of a computer

Description automatically generated with low confidence

* 1. ioPrint(string, newLine=False): This function is used to print text to the user in the console printer. Arguments of this function are [string] string, the text to print , [newLine] boolean, whether the print should be in its own line. This function will get the printer widget, enable editing to write changes, get the last character of the text, start a new line if the last string didn't end with one, print the text, disable editing and update text.
  2. ioKey(): This function get a character from the console keyboard and returns the string. This gets the full string of the keyboard widget (the last character is always '\n' even when widget is empty, so we ignore that character).
  3. ioInstExec(instruction): This function is used for IN, OUT and CHK operations based on opcode. If it is IN instruction, get the value of the first character from Keyboard (in case of overflow, truncate the bits). If it is OUT instruction, get the value from the register then convert the value to a character and send the text to the printer. If it is CHK instruction, then it used to check device status to register.