Design note

Project Summary

The Phase 1 of the simulator mainly focus on building these following parts

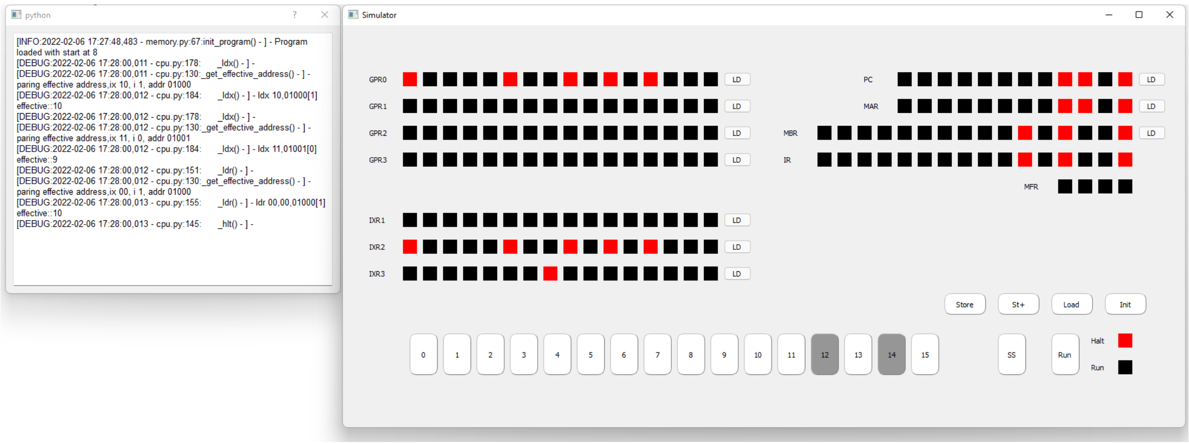
- The GUI
- The main simulator framework
- The memory
- registers
- Certain instructions
  - HLT
  - LDR
  - STR
  - LDA
  - LDX
  - STX

Project structure

```
├── LICENSE
├── README.md
├── document
│   └── project1_planning.md
├── project
│   └── src
│       ├── IPL.txt          //the IPL file for loading program
│       ├── __init__.py
│       ├── cache.py         //Placeholder for caches, to implant in phase2
│       ├── constants.py     //Define constants that would be used across the project
│       ├── cpu.py           //Define the cpu Class, the main simulater logic happens here
│       ├── memory.py        //Define the memeory class
│       ├── mfr.py           //predifined mfr errors, to be used in phase3
│       ├── op_code_list.py  //a map of all op_codes
│       ├── register.py      //Define the register class, used to initiate the registers
│       ├── simulator_GUI.py //The entrance of code, as well as the GUI codes
│       └── word.py          //Define the word Class, which is used to hold data in both memory and register
├── requirements.txt
├── setup.py
└── tests
    ├── IPL.txt
    └── test_utils.py       //test functions to run against backend codes.
```

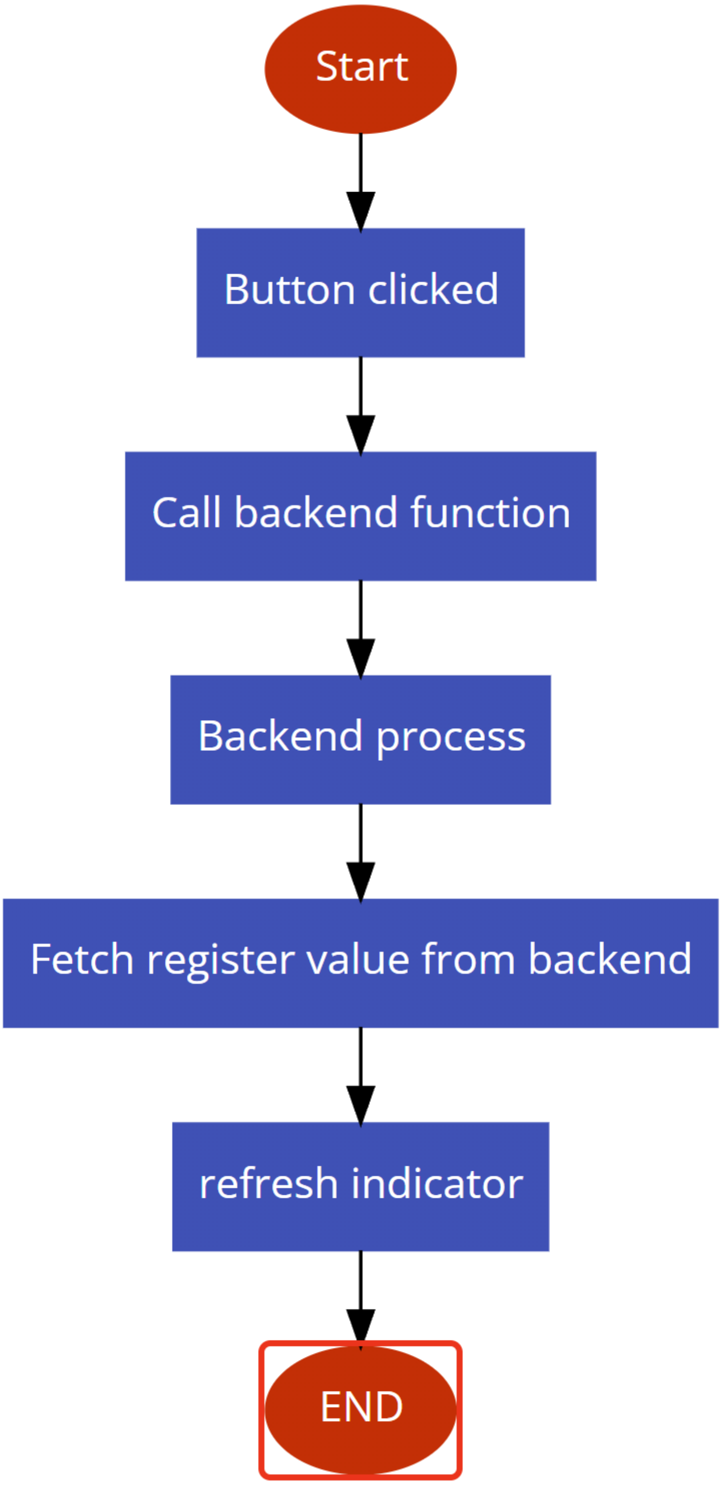
The GUI

The GUI is developed with the PyQT5 framework



```
[INFO:2022-02-06 17:27:48,483 - memory.py:67:init_program() - ] - Program
loaded with start at 8
[DEBUG:2022-02-06 17:28:00,011 - cpu.py:178:      _ldx() - ] -
[DEBUG:2022-02-06 17:28:00,011 - cpu.py:130:_get_effective_address() - ] -
paring effective address,ix 10, i 1, addr 01000
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:184:      _ldx() - ] - ldx 10,01000[1]
effective::10
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:178:      _ldx() - ] -
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:130:_get_effective_address() - ] -
paring effective address,ix 11, i 0, addr 01001
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:184:      _ldx() - ] - ldx 11,01001[0]
effective::9
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:151:      _ldr() - ] -
[DEBUG:2022-02-06 17:28:00,012 - cpu.py:130:_get_effective_address() - ] -
paring effective address,ix 00, i 1, addr 01000
[DEBUG:2022-02-06 17:28:00,013 - cpu.py:155:      _ldr() - ] - ldr 00,00,01000[1]
effective::10
[DEBUG:2022-02-06 17:28:00,013 - cpu.py:145:      _hlt() - ] -
```

- As can be seen above, the main GUI consists two parts, the simulator interface and the debug console.
- For the simulator part, I've done certain abstractions on the register indicators and the buttons.
  - Each line of register indicator is generated by the class `RegisterGUI` . This class also provides a method `refresh_label` allowing us to simply using binary string to refresh the indicator.
  - Each button is generated by class `PressButton`. This class also provides a method `on_click` to bind corresponding method calls.
  - When `LD` button is pressed, the value on the switch will be fetched and stored into the corresponding register. After this progress completes, the GUI will fetch result from the backend and refresh the indicator.
  - All other button has the same logic: call corresponding backend function, and refresh the indicator upon finish.
  -



- Abstraction of register indicators: The register indicators could be represented by the following python dictionary.

```python
map_reg_location = {
    # name: x_location,y_location,reg_count,button_function,has_button
    "GPR0": [40, 70, 16, cpu_instance.gpr[0].set, True],
    "GPR1": [40, 110, 16, cpu_instance.gpr[1].set, True],
    "GPR2": [40, 150, 16, cpu_instance.gpr[2].set, True],
    "GPR3": [40, 190, 16, cpu_instance.gpr[3].set, True],
    "IXR1": [40, 280, 16, cpu_instance.ixr[1].set, True],
    "IXR2": [40, 320, 16, cpu_instance.ixr[2].set, True],
    "IXR3": [40, 360, 16, cpu_instance.ixr[3].set, True],
    "PC": [780, 70, 12, cpu_instance.pc.set, True],
    "MAR": [780, 110, 12, cpu_instance.mar.set, True],
    "MBR": [660, 150, 16, cpu_instance.mbr.set, True],
    "IR": [660, 190, 16, cpu_instance.ir.set, False],
    "MFR": [1020, 230, 4, cpu_instance.mfr.set, False],
}
```

- For the console log part, it mainly used `QTextEditLogger` from Pyqt5 to serve as a python log handler. This log box will catch every log the simulator program generated.

The main simulator framework

- The main simulator framework is developed in `cpu.py` for class `CPU`.
    - Table of data structure in class `CPU`

data	usage
memory	the memory
pc	the pc register
mar	the mar register
mbr	the mbr register
gpr[]	the gpr register in a list
ixr[]	the ixr register in a list
cc	the cc register place holder
mfr	the mfr register
ir	the ir register
halt_signal	to indicate if halt or not

-
    - Table of method in class `CPU`

method	usage
init	used to init cpu instance, assigning registers and memory to cpu.
run	used by run button in GUI, to run the program until halt_signal
run_single_cycle	used by SS button in GUI, to run a single instruction
store	used by store button in GUI, to store mbr into memory[mar]
store_plus	used by ST+ button in GUI, to store mbr and add 1 in mar
load	used by load button in GUI, to load memory[mar] to mbr
get_all_reg	return all register status, used to refresh register indicators in GUI
_get_func_by_op	return specific method to be executed corresponding to the op_code
_get_effective_address	return the effective address according to ix， i, addr value
_hlt	the method to be executed in hlt op_code
_str	the method to be executed in str op_code
_lda	the method to be executed in lda op_code
_ldx	the method to be executed in ldx op_code
_stx	the method to be executed in stx op_code
_ldr	the method to be executed in ldr op_code

- The main loop(single step)
  - mar = pc
  - mbr = memory[mar]
  - ir = mbr
  - call _get_func_by_op() to get the specific function
  - if halt_signal -> return
  - pc.add(1)

memory

- Memory is implemented in `memory.py` for class `Memory`
  - Table of data structure in class `Memory`

data	usage
memroy[]	used to contain data
size	represent the size of memory

- Table of method in class `Memory`

method	usage
validate_addr	used to determine if the address is valid, will trigger `MemReserveErr` or `MemOverflowErr` if illegal
reset	reset all memory to 0, used by pressing button init
store(address,value)	store value to address in memory
store_reserved(target,value)	store value to reserved locations
load(address)	return memory[address]
init_program(file_path)	read from `file_path` and preload the program into memory

register

- register is implemented in `register.py` for class `Register`
  - Table of data structure in class `Register`
data	usage
value	used to contain data
max	represent the max size of register, will raise a exception if value > max
  - Table of method in class `Register`

method	usage
init	initiate the register instance
validate	check if the value has exceeded the max value of register
set(value)	set the value of the register
get	return the value of the register
reset	set register to 0, used by pressing button init
add(value)	add certain value to register, mainly used by `self.pc.add(1)` and `self.mar.add(1)`
rotate(lr,al,count)	register rotate placeholder to be implemented in the future
shift(lr,al,count)	register shift placeholder to be implemented in the future

Instructions

_hlt	the method to be executed in hlt op_code
_str	the method to be executed in str op_code
_lda	the method to be executed in lda op_code
_ldx	the method to be executed in ldx op_code
_stx	the method to be executed in stx op_code
_ldr	the method to be executed in ldr op_code