The University of Hong Kong

COMP4801 Final Year Project

# Final Report

## Simulation of Simple Computer System for Teaching

Wong Yin Lok
3035094933

13 April 2017

## Abstract

This Final Year Project – Simulation of Simple Computer System for Teaching is about the development of a simulator for the purpose of enhancing teaching and learning experience in the Computer Science course COMP2120 Computer Organization. This project has completed with the delivery of two simulation components in one simulator program, one for CPU simulation and the other for cache memory simulation. Comparing to the simulators that have been developed for this course, the deliverable from this project offers an equally comprehensive yet more intuitive to use simulator program with added features. Users will be able to define their custom instructions and see the simulation result via the dynamic display, together with the cache memory simulation that presents to them cache operations in practice. In the process of using this simulator, users will consolidate their relevant knowledge to a practical level and at their own pace.

## Acknowledgement

This project is supervised by Dr. K. P. Chan from the Department of Computer Science at the University of Hong Kong.

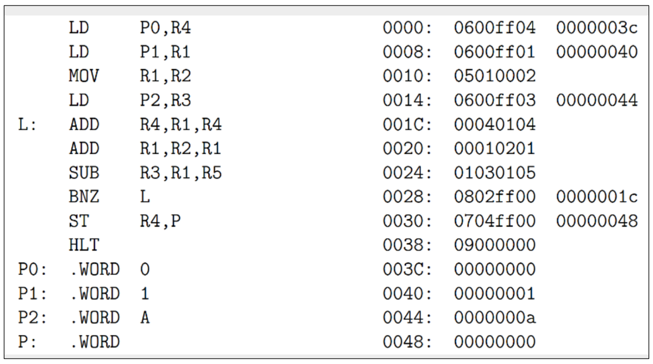
## Table of Content

## List of Figures

## List of Tables

## Background

Computer organization and architecture is the basic knowledge to be learnt, if not mastered, in Computer Science and related studies. The study of computer organization explores the fundamental functions performed by any computer system – to move data around and perform simple arithmetic operations. These two simple functions, however, are not very intuitive to be programmed. As the operations are carried out by the CPU, which identifies only bits of 1 and 0, all the instructions to be programmed have to be translated to segments of numbers - known as the assembly code - to indicate specific operations and data location according to a scheme referred to as the instruction set. Figure 1 shows an example programme of CPU execution.



```
     LD    P0,R4         0000:  0600ff04  0000003c
     LD    P1,R1         0008:  0600ff01  00000040
     MOV   R1,R2         0010:  05010002
     LD    P2,R3         0014:  0600ff03  00000044
L:   ADD   R4,R1,R4      001C:  00040104
     ADD   R1,R2,R1      0020:  00010201
     SUB   R3,R1,R5      0024:  01030105
     BNZ   L             0028:  0802ff00  0000001c
     ST    R4,P          0030:  0704ff00  00000048
     HLT                 0038:  09000000
P0:  .WORD 0             003C:  00000000
P1:  .WORD 1             0040:  00000001
P2:  .WORD A             0044:  0000000a
P:   .WORD               0048:  00000000
```

Figure 1. example programme of CPU execution

The instruction set is a set of mnemonic instructions, including common ones like ADD, SUB and MOV that is performed by the CPU. The disassembled, pseudo code is written in a form like “ADD R4,R1,R4” while the actual assembly code perceived by the CPU is like series of hexadecimal numbers correspond-ing to different operations and registers or memory addresses. Simply understanding the disassembled code but not being able to translate them into hexadecimal assembly codes is not enough. Students do need to understand the translation.

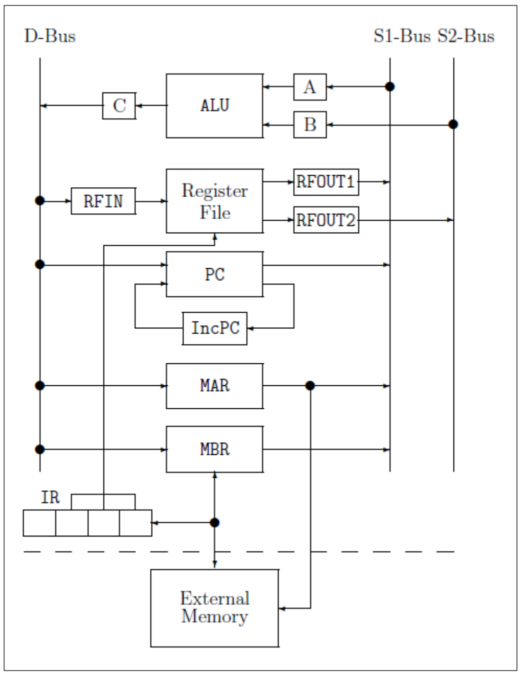
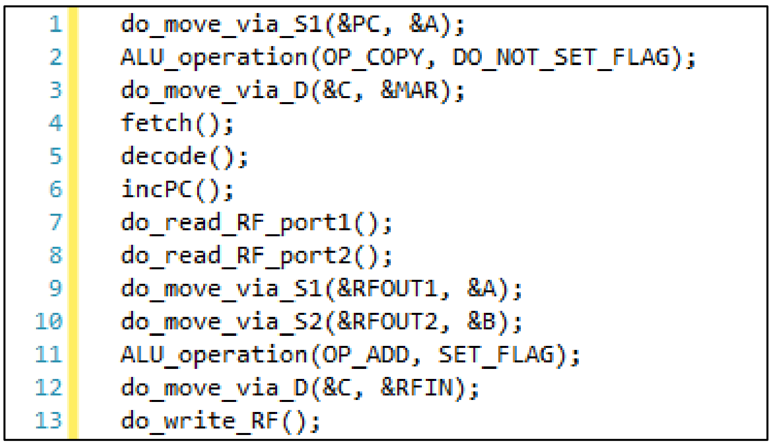


Figure 2. CPU architecture used in COMP2120

This means students have to write code in series of digits and try to keep track of the actual data movements and operations based on these abstract lines of hexadecimal operation codes. Not only is the code written in hexadecimal numbers difficult to follow, the thorough understanding of actual CPU operations is not reflected in the code.

Figure 2 shows an example CPU architecture used in the Computer Science course Computer Organization. In actual CPU operations, each operation code in the instruction set, e.g. ADD and SUB, is implemented by a series of interactions between the CPU components.
For example, in an ADD operation, internal register data are read from the RF ports and moved via the system buses, then followed by some other actions involving other CPU components, as shown in figure 3. These details are not shown or practiced when students write codes, since the programme codes will only be lines of instruction code like “00040104”

representing "ADD R4,R1,R4" without disclosing the actions of the CPU components required for that single line of code.



```
do_move_via_S1(&PC, &A);
ALU_operation(OP_COPY, DO_NOT_SET_FLAG);
do_move_via_D(&C, &MAR);
fetch();
decode();
incPC();
do_read_RF_port1();
do_read_RF_port2();
do_move_via_S1(&RFOUT1, &A);
do_move_via_S2(&RFOUT2, &B);
ALU_operation(OP_ADD, SET_FLAG);
do_move_via_D(&C, &RFIN);
do_write_RF();
```

Figure 3. sample "ADD" definition translated from the implementation in C of the simulator[1] currently used in COMP2120

In addition, memory cache replacement can be a confusing process in which different slots of the memory cache are swapped out and replaced by new memory content. Memory caching is the process of reading comparably slow memory into fast-access CPU cache. While the number of slots and the size of the memory cache are fixed, the replacements come in different orders, targeting different slots every time depending on the replacement scheme. This is, as what we see with CPU operations, difficult to follow without proper visualization. To help students better picture the above details, the idea of a graphical simulator was born.

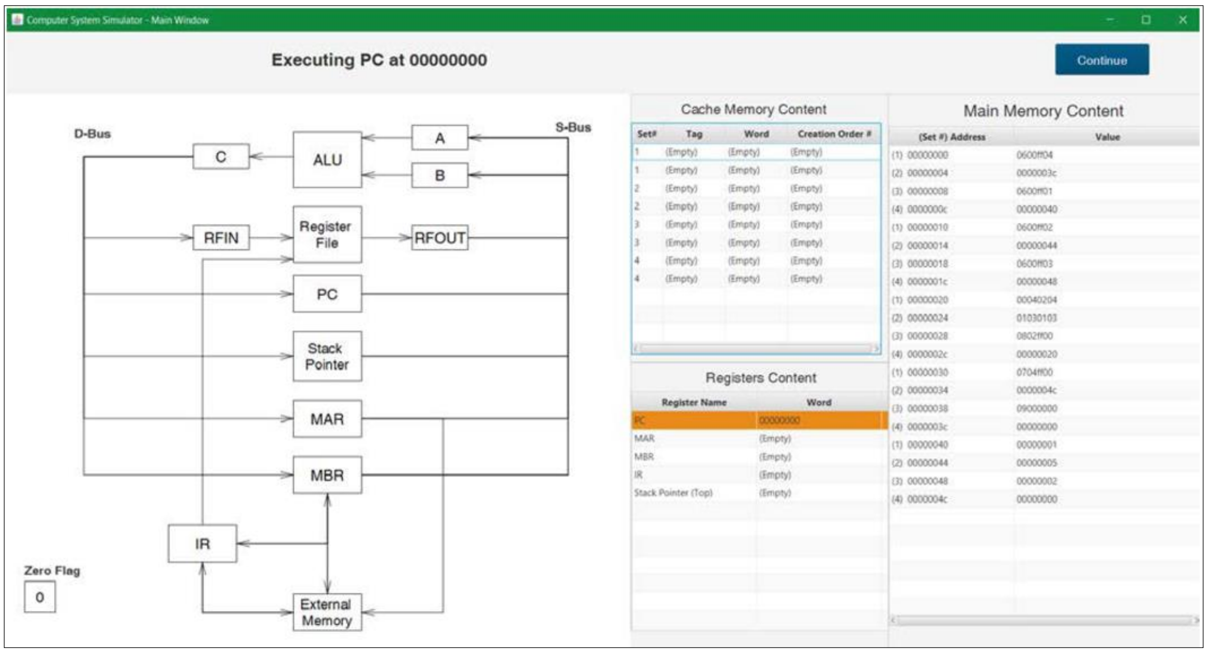
## Objective

The grand objective of this project is to improve teaching and learning in the course Computer Organization regarding the working mechanism of CPU and cache memory. More specifically, in the CPU simulation, it is to help students visualize and understand the detailed operations inside the CPU for different instructions; in cache memory simulation, it is to help students understand the logic of different memory replacement scheme by visualizing the update of memory content.

## Previous Work

This project is not new. Last year, another student took up this project as his Final Year Project. But the Java simulator built in that project turned out to be more of a step by step "video player" of execution statements loaded from some configuration files. The CPU simulation would read in a file containing the programme executions and display the execution results of every line of code one after another in a description panel controlled by the user, and the memory panel would display the update of memory in respective memory slots. Figure 4 shows a screenshot of the simulator from the project last year.

[1] sim.cc. Hong Kong: HKU COMP2120



Cache Memory Content

| Set# | Tag | Word | Creation Order # |
|---|---|---|---|
| 1 | (Empty) | (Empty) | (Empty) |
| 1 | (Empty) | (Empty) | (Empty) |
| 2 | (Empty) | (Empty) | (Empty) |
| 2 | (Empty) | (Empty) | (Empty) |
| 3 | (Empty) | (Empty) | (Empty) |
| 3 | (Empty) | (Empty) | (Empty) |
| 4 | (Empty) | (Empty) | (Empty) |
| 4 | (Empty) | (Empty) | (Empty) |

Registers Content

| Register Name | Word |
|---|---|
| PC | 00000000 |
| MAR | (Empty) |
| MBR | (Empty) |
| IR | (Empty) |
| Stack Pointer (Top) | (Empty) |

Main Memory Content

| (Set #) Address | Value |
|---|---|
| (1) 00000000 | 0600ff04 |
| (2) 00000004 | 0000003c |
| (3) 00000008 | 0600ff01 |
| (4) 0000000c | 00000040 |
| (1) 00000010 | 0600ff02 |
| (2) 00000014 | 00000044 |
| (3) 00000018 | 0600ff03 |
| (4) 0000001c | 00000048 |
| (1) 00000020 | 00040204 |
| (2) 00000024 | 01030103 |
| (3) 00000028 | 0802ff00 |
| (4) 0000002c | 00000020 |
| (1) 00000030 | 0704ff00 |
| (2) 00000034 | 0000004c |
| (3) 00000038 | 09000000 |
| (4) 0000003c | 00000000 |
| (1) 00000040 | 00000001 |
| (2) 00000044 | 00000005 |
| (3) 00000048 | 00000002 |
| (4) 0000004c | 00000000 |

Figure 4. screenshot of the simulator from previous project, captured from the corresponding final report[2]

While the memory simulation might be dynamic and representable enough, the CPU simulation was lacking in strengthening students' understanding of the underlying CPU actions that are performed behind each instruction.

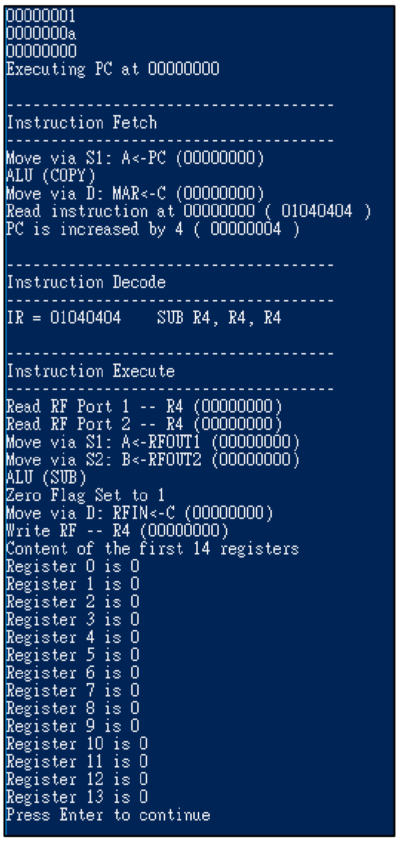
The graphical display of the simulator would highlight the CPU component involved according to the sequence of program executions. However, the relationship between individual CPU actions and the corresponding instruction was not clearly demonstrated. In addition, visualization alone has only minimal effect to the understanding of behind-the-scene CPU operations, but thorough understanding could hardly be achieved just by watching tens of seconds of animation.

The key is to introduce the element of practical implementation, so as to install knowledge via solid practice and hands-on experience. This current project, therefore, basically started all over again from scratch in order to introduce the new element of custom instruction definition, which will be mentioned in later parts of this report and largely deviated from the approach adopted in the previous project. For cache memory simulation, despite the fact that it was probably catered fairly in the project last year, its development in this current project depended heavily on the CPU simulation where the source codes of the two simulations interconnect. Thus, in the end, no code from the previous project was taken as reference or recycled throughout the course of this project.

There is, however, another simulator[3] of an older origin, written in C and has been used as the tool in the course for several years, from that this project has taken reference from. This simulator, despite being comprehensive in functionalities, shares the same downsides as the other simulator.

[2] Wong, Jing Hing(Kent). FINAL REPORT TOPIC: COMPUTER SYSTEM SIMULATOR. Hong Kong: 2015

[3] sim.cc. Hong Kong: HKU COMP2120



```
00000001
0000000a
00000000
Executing PC at 00000000

------------------------------------
Instruction Fetch
------------------------------------
Move via S1: A<-PC (00000000)
ALU (COPY)
Move via D: MAR<-C (00000000)
Read instruction at 00000000 ( 01040404 )
PC is increased by 4 ( 00000004 )

------------------------------------
Instruction Decode
------------------------------------
IR = 01040404    SUB R4, R4, R4

------------------------------------
Instruction Execute
------------------------------------
Read RF Port 1 -- R4 (00000000)
Read RF Port 2 -- R4 (00000000)
Move via S1: A<-RFOUT1 (00000000)
Move via S2: B<-RFOUT2 (00000000)
ALU (SUB)
Zero Flag Set to 1
Move via D: RFIN<-C (00000000)
Write RF -- R4 (00000000)
Content of the first 14 registers
Register 0 is 0
Register 1 is 0
Register 2 is 0
Register 3 is 0
Register 4 is 0
Register 5 is 0
Register 6 is 0
Register 7 is 0
Register 8 is 0
Register 9 is 0
Register 10 is 0
Register 11 is 0
Register 12 is 0
Register 13 is 0
Press Enter to continue
```

Figure 5. screenshot of running the simulation from the simulator currently used in COMP2120

As shown in figure 5, the simulator is a command-line-based program that runs on the terminal interface with text output. It is rigid and inflexible in that all instructions used in the programme are predefined in the simulator logic and what users do are only supplying an external configuration file and then reading the text output from the console window.

The simulator from the final year project last year attempted to improve on this simulator by providing more vivid graphical content, yet the core of the problem remains that students do not easily gain a thorough understanding to the underlying hardware-level operations by simply reading from the simulation output.

This current project tackles specifically on this problem by introducing new features, while taking reference from the implementation logics of the standard operations defined in this simulator currently used in the course.

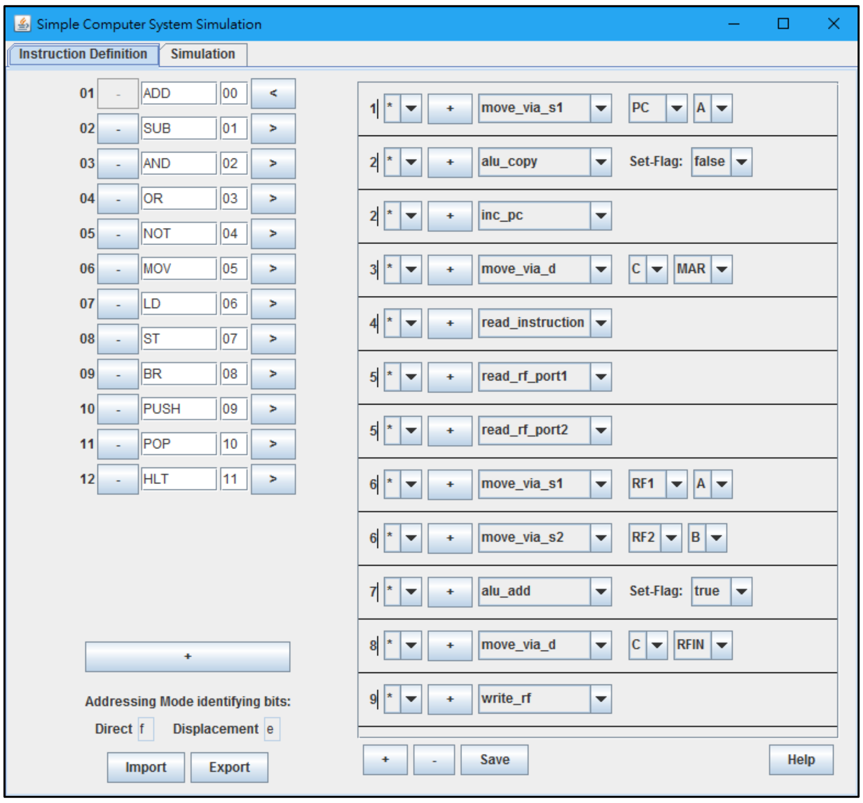
## Scope and Deliverables

The final deliverable is a Java program including two simulation components, namely the CPU simulation and the cache memory simulation. Java was chosen because of the hopeful reusability of previous codes and the platform independent nature of the compiled program which can then be distributed to students as an educational tool. While simulation of any of these two system components can involve a wide range of relevant concepts and complex implementation details, the scope of this project is carefully controlled in order to streamline the deliverable and make it focused on its educational purpose.

### Instruction Definition

To begin with, the CPU simulation contains two major parts implemented in two separate display panel.

The first part is mnemonic instruction definition, and its completed interface is shown in figure 6. As stated in previous sections, the objective of the CPU simulation is to allow users to learn and better understand the actual operations happening inside the CPU, via practical hands-on experience rather than merely watching the graphical display run. In this part of the simulation, users will be able to define the implementations of custom instructions.



| | | | | |
|---|---|---|---|---|
| 01 | - | ADD | 00 | < |
| 02 | - | SUB | 01 | > |
| 03 | - | AND | 02 | > |
| 04 | - | OR | 03 | > |
| 05 | - | NOT | 04 | > |
| 06 | - | MOV | 05 | > |
| 07 | - | LD | 06 | > |
| 08 | - | ST | 07 | > |
| 09 | - | BR | 08 | > |
| 10 | - | PUSH | 09 | > |
| 11 | - | POP | 10 | > |
| 12 | - | HLT | 11 | > |

Addressing Mode identifying bits:
Direct f Displacement e

| | | | | |
|---|---|---|---|---|
| 1 | * | move_via_s1 | PC | A |
| 2 | * | alu_copy | Set-Flag: | false |
| 2 | * | inc_pc | | |
| 3 | * | move_via_d | C | MAR |
| 4 | * | read_instruction | | |
| 5 | * | read_rf_port1 | | |
| 5 | * | read_rf_port2 | | |
| 6 | * | move_via_s1 | RF1 | A |
| 6 | * | move_via_s2 | RF2 | B |
| 7 | * | alu_add | Set-Flag: | true |
| 8 | * | move_via_d | C | RFIN |
| 9 | * | write_rf | | |

Figure 6. interface for mnemonic mapping of the CPU simulation

As seen from figure 6, the input of the mnemonic names is on the left of the panel where users will add and type in the name of the mnemonic instructions that they want to define, with the hexadecimal operation code for that instruction. For the sake of maintaining the program logic and reducing its complexity and thus the possibility of errors, the number of instructions that can be defined is capped at a maximum of 16, which should be more than enough for writing general programmes.

At the start of the simulator program, the left of the panel will be initialized with one single entry of empty textfields and buttons, shown in figure 7.

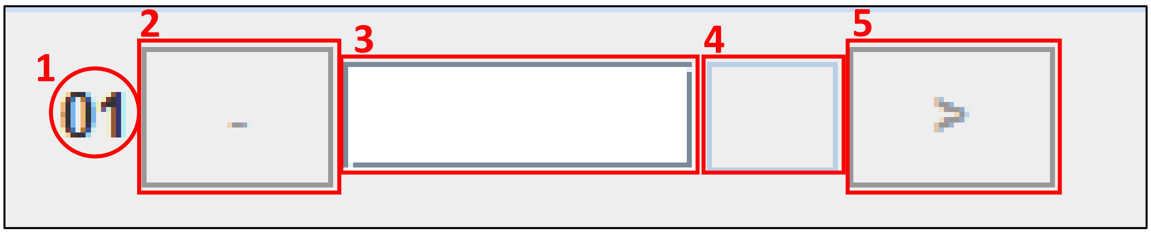


Figure 7. dissection of an empty instruction entry

Section labelled number 1 in figure 7 is the graphical index of the instruction entry. The button labelled numbered 2 is the removal button that is not enabled for the first entry but on all others for removal of the particular instruction. This suggests that the simulator works with at least one instruction definition and cannot function with no instruction input. Textfields numbered 3 and 4 are respectively the entry for the mnemonic symbol of the instruction and the hexadecimal operation code. The length of the mnemonic symbol is capped at six characters while that of the opcode is capped at two numeric characters due to the fact that the opcode is signified by two bits in an eight-bit instruction word in hexadecimal assembly code. After the input of the mnemonic symbol is captured, the opcode textfield and the button on the right labelled 5 will be enabled to allow the expansion of the right panel for configuring hardware-level actions that the instruction will perform. This is to ensure that every instruction will get its mnemonic symbol and opcode defined properly.

On the right of the CPU simulation panel as shown in figure 6, corresponding CPU actions that implement the desired function of the custom instruction can be added and selected in the drop-down boxes. One rectangular entry represents an action to be performed in an execution timeframe of the CPU. There are three types of layout for the action entries, as shown in figure 8.1, 8.2 and 8.3.

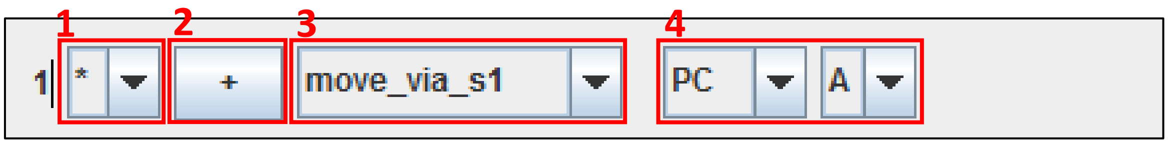


Figure 8.1. timeframe layout for system bus-related actions

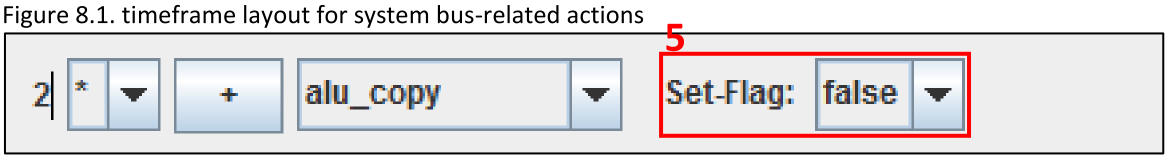


Figure 8.2. timeframe layout for ALU operations

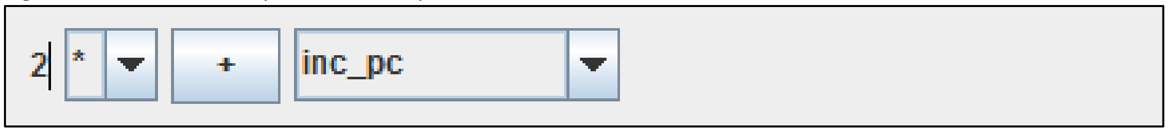


Figure 8.3. timeframe layout for all other actions

Sections labelled number 1 to 3 in figure 8.1 are commonly seen in all three types of layout. The drop-down box numbered 1 determines the addressing mode of that particular CPU action, which will be discussed later. The button numbered 2 adds an additional CPU action to the current timeframe for the purpose of parallelizing the operations, which will also be discussed later in this section. The drop-down box numbered 3 determines the CPU action to be performed, the list of CPU actions - a total of twenty-two different actions - supported by this simulator is shown as table 1 and will be discussed further in this section. The two drop-down boxes in section number 4 in figure 8.1 is specific to entries when the CPU action selected in drop-down box number 3 is related to data movement via the system buses, i.e. “move_via_s1”, “move_via_s2” and “move_via_d”, where the first drop-down box on the left specifies the data source while the right box specifies the destination of the data movement.

Figure 8.2 shows the layout when the CPU action involves operations by the ALU unit, e.g. “alu_add” and “alu_not”. The section labelled number 5 determines whether the result of the ALU operation should update the zero-flag on which the “branch” action depends. If Set-Flag is true, when the result of the ALU operation is not zero, zero-flag is set to 0, otherwise set to 1; if Set-Flag is false, the result of the ALU operation will not update the zero-flag.

Figure 8.3 shows the layout when the CPU action involves operations other than the two types mentioned above. There will be no supplementary information needed for these other actions. Note that these differences in the information needed for different CPU actions are important, not only to properly implement the desired behaviour of the action in the simulation, but also in the import and export functionalities provided by the simulator, which will be covered later in this section.

| Elementary Actions |
|---|
| move_via_s1 |
| move_via_s2 |
| move_via_d |
| inc_pc |
| read_rf_port1 |
| read_rf_port2 |
| write_rf |
| alu_add |
| alu_sub |
| alu_and |
| alu_or |
| alu_not |
| alu_copy |
| read_instruction |
| read_memory |
| write_memory |
| branch |
| dec_dp |
| inc_sp |
| mar_to_temp |
| temp_to_mar |
| halt |

Table 1. list of hardware-level actions supported by this simulator

Besides the add button mentioned above, i.e. button numbered 2 in figure 8.1, as seen from figure 6, there is another add button alongside other buttons near the bottom of the right of the simulation panel. This difference in functionality of this additional add button with all the other add buttons within the timeframe entries is worth noting as well as the significance of the index numbers in the timeframes. For example as seen from figure 8.2 and 8.3, the two entries, as part of the definition for the custom instruction “ADD” as shown in figure 6, share the same graphical index number. These are designed for parallelization. As different CPU operations involve the use of different resources, there are times when the operations are independent to each other in terms of both time and resources that they can be parallelized instead of having to proceed in a serial manner. In this case, when “alu_copy” is being performed, “inc_pc” can be performed simultaneously without causing any abnormality of the programme logic or resource conflict. Hence, after the user have selected “alu_copy” in timeframe 2, the user can then click the add button within the entry to initialize another parallel action within the same timeframe. If the user hopes to initialize a new timeframe in a sequential manner, e.g. from index 1 proceeding to index 2, the user can then click the add button at the bottom.

The “-” button removes only the last timeframe with all its action entries, for the sake of implementation simplicity - as the maintenance of the proper ordering of the arraylist of the entries with some sharing the same graphical index is counter-intuitive enough, not to mention the need to cater for removal of items in the middle of the list. That means if the last timeframe, for example, three action entries all with the same graphical index, the three entries will all be removed when the removal button is clicked.

The “Save” button, together with the “+” and “-” button at the bottom, will only be enabled when there is an instruction expanded for action configuration, i.e. when the button numbered 5 in figure 7 of the corresponding instruction shows “<”, indicating that it is currently being edited. The “Save” button is used to save the custom definition for the use of CPU simulation. The meaning of “saving the definition” will be explained in the Methodology section.

Similar to the limit in instructions, there is a limit of thirty action entries that can be added for each instruction, regardless of the index number, parallel or sequential. This number should be more than enough for definition of standard instructions.

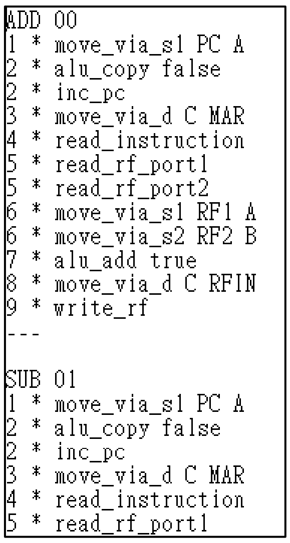
After an instruction is configured and saved, after are some other actions the user can perform before going into the actual simulation. For example, by clicking the rectangular “+” button at the bottom left, the user can then add another instruction entry for a maximum total of sixteen instructions.

The textfields under "Addressing Mode identifying bits" specifies the hexadecimal bits that represent the use of different addressing mode. This simulator supports by default two addressing modes, with "f" representing the use of direct addressing and "e" representing the use of displacement addressing. These bits correspond to selection from the drop-down box numbered 1 shown in figure 8.1.

If the user chooses "*" from the drop-down box, that particular action will be executed regardless of which addressing mode the current instruction is in, as specified in the programme input, which will be covered in the next subsection Graphical Simulation. If the user chooses "f" or "e", that particular action will only be executed when the addressing mode specified is in the corresponding direct or displacement addressing.

There are at last the import and export buttons. This simulator provides no internal storage mechanism but instead provides the import and export function in consideration of possible distribution and submission of the definition text files in the course being easier than each of the students having their own saved state of the simulator.

After the definition is saved, the user can click the export button to export a definition text file containing the definition of all saved instructions. A section of a sample text file is shown in figure 9, while a complete definition file containing 12 standard instructions can be viewed in appendix 1.



```
ADD 00
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_add true
8 * move_via_d C RFIN
9 * write_rf
---

SUB 01
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
```

Figure 9. extract of sample definition text file

The exported definition will be in .txt format. And the definitions saved in the text file can be loaded to the simulator by clicking the import button and selecting the text file.

Figure 9 provides a clue to the format of the definition file. For each instruction definition, the syntax of the file begins with a line containing the instruction mnemonic symbol and the two-bits opcode, separated by space. Following the first line, the actions in the instruction are specified by lines starting with the timeframe index, followed by the addressing mode identifying bit, the hardware-level action string, then the optional information depending on the action, all separated by space. At the end of the instruction, a line of three dashes and another empty line is used to signal the end of the definition and the start of another definition can then begin.

The definition text file can be exported or composed manually. But upon manually construction, the syntax of the content should follow the-above-mentioned. Otherwise, the import may fail. Further description about handling mal-formatted import is covered in section Testing and Evaluation.

## Graphical Simulation

The second part of the CPU simulation is animated graphical display. This graphical display is essentially a video player that shows data movements and operations in the CPU. Figure 10 shows the completed interface of the graphical display.

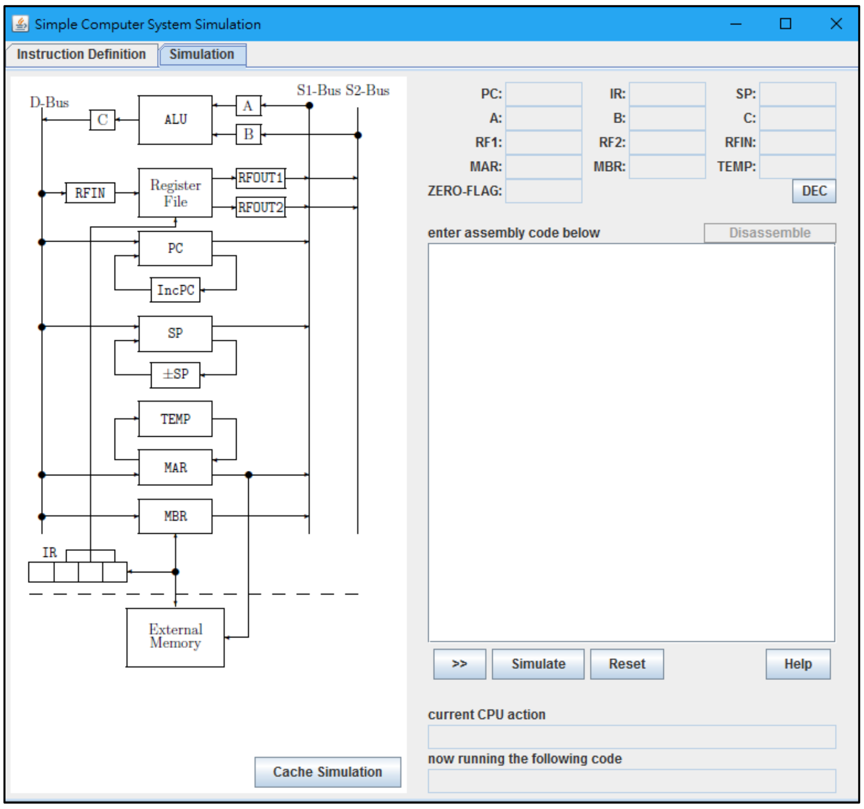
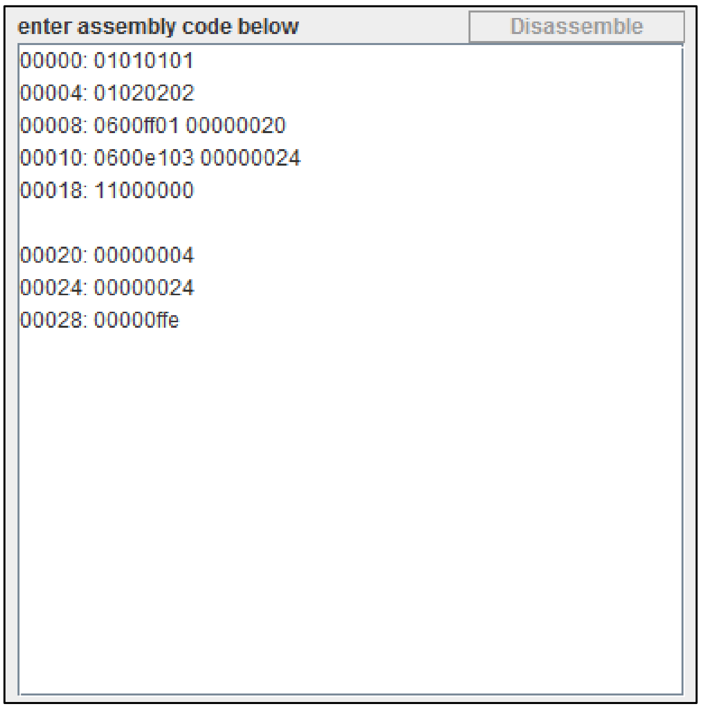


Figure 10. interface for graphic display tab of the CPU simulation

The simulator provides a large text field for users to type in their programme in hexadecimal assembly codes. The operation codes are mapped to their corresponding mnemonic instructions in the previous stage. This programme input serves as the basis for the graphical display to run.

An example assembly programme in hexadecimal codes is shown in figure 11. Note that there are certain syntax that the programme has to follow in order for the simulator to correctly load and decode the code lines.



```
00000: 01010101
00004: 01020202
00008: 0600ff01 00000020
00010: 0600e103 00000024
00018: 11000000

00020: 00000004
00024: 00000024
00028: 00000ffe
```

Figure 11. sample programme input

Every line of code should begin a with a 5 bits address. The entire programme is essentially assigning different data into different memory address slots. Following the 5 bits has to be a colon and a space. Then comes the content to be assigned to the address. The data content can take two different formats, one being a one-word instruction/data, the other being a two-word instruction/data. The former takes in an 8 bits data string while the latter takes in a 8 bits data string, followed by a space separator, followed by another 8 bits data string. These syntax rule has to be strictly followed, otherwise, an error message will pop up and the simulation will not begin. Details on handling mal-formed input will be discussed in the section Testing and Evaluation.

When the “Simulate” button is clicked, there will be colored points, alongside the data content in small text, representing data signals, moving on the CPU image to indicate data movement. Relevant CPU hardware components on the path, e.g. PC and MAR, will have content update which will be shown in the respective description fields at the top right beside the display.

There is a “DEC” button at the top right near the hardware description fields, which serves the purpose of changing the numeric base of the display content. As data in the simulation is represented in hexadecimal numbers, which is not intuitive to read, the simulator offers another decimal base of representation. The bases of numeric representation can be changed back and forth between decimal and hexadecimal anytime upon the user’s discretion. Note that the change to decimal base is only to provide the user with an optional, and more convenient visual representation, and is independent of the data content concerned. With this said, all description fields - except PC, MAR and SP, which are always containing address data in hexadecimal representation, and the ZERO-FLAG which can only be in 1 or 0 - will have their bases changed when the button is clicked, even when address data in hexadecimal get into the temporary C register.

When the “Simulate” button is clicked, the simulation begins by reading the content in address “00000” as specified in the programme input. The “Simulate” button then shows “Pause”, and when clicked, will pause the simulation to allow the user to read and digest the information on the graphical display or the descriptions fields on the right. After clicking “Pause” and the simulation paused, the button shows “Simulate” again and can resume the simulation when clicked again.

The double-arrow button beside the “Simulate” button controls the speed of the simulation animation. There are three different speeds for the simulation that the user can switch from one to another using the speed button during the simulation. This design is to allow users to quickly go through the simulated actions which have been seen repeatedly under many occasions that users are already familiar with. The speed will be reset automatically to the default pace upon entering a new line of instruction to ensure that no instruction simulation is skipped unintentionally.

The “Reset” button stops the simulation if it is running and reset all the description fields and internal variables used in the simulation.

The two textfields at the bottom right are descriptions to the current simulation status, providing textual hints for users to follow the simulation. The description field on top displays one or more of the twenty-two elementary CPU actions listed in table 1 that are currently running, and the bottom field shows the eight-bits instruction word that is being executed together with the address containing that word.

As students are required to programme with custom instruction sets in the course, this part of the simulation allows them to test their very own implementations and enhance understanding of the principle. However, it is not possible for the simulation to support definitions of any unseen operation, because after all, the functions of all the custom instructions are limited to the combination of the hardware-level actions are listed in table 1.

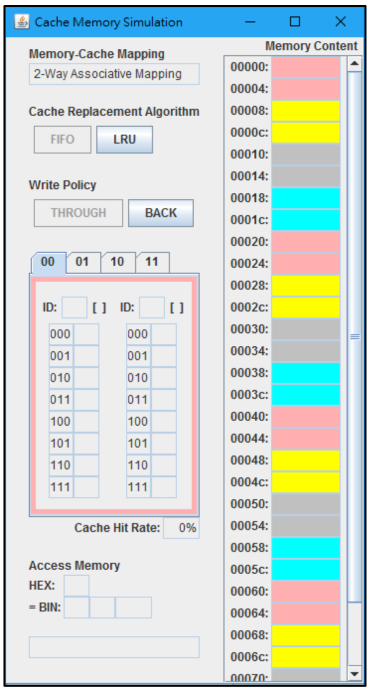
| Guaranteed Instructions |
|---|
| ADD |
| SUB |
| AND |
| OR |
| NOT |
| MOV |
| LD |
| ST |
| BR/BZ/BNZ |
| PUSH |
| POP |
| HLT |

Table 2. Guaranteed instruction definitions by this simulator

For example, for operations that require the use of ALU unit to perform simple arithmetic like addition and subtraction, the ALU operations will have to be pre-defined by the simulation system. Therefore, this simulator will only guarantee that the definitions of twelve standard instructions can be achieved, as listed in table 2, with possibility of defining other novel instructions. The sample definition of the standard instructions is attached in appendix 1.

Note that the "HLT" instruction is not a genuine instruction in a sense that it does not require the step of instruction fetch, its functionality of halting the running simulation is simply by the power of the single "halt" action in the definition. This "halt" action is designed only for the purpose of the simulator. As in real-life CPU execution, the cycle is essentially an infinite loop that does not stop at any point. In contrary to that, the simulation here better has a finite stop to signal the user of a completion. Therefore, this "halt" action and thus "HLT" instruction are added.

## Cache Simulation


Cache Memory Simulation

Memory-Cache Mapping: 2-Way Associative Mapping

Cache Replacement Algorithm: FIFO LRU

Write Policy: THROUGH BACK

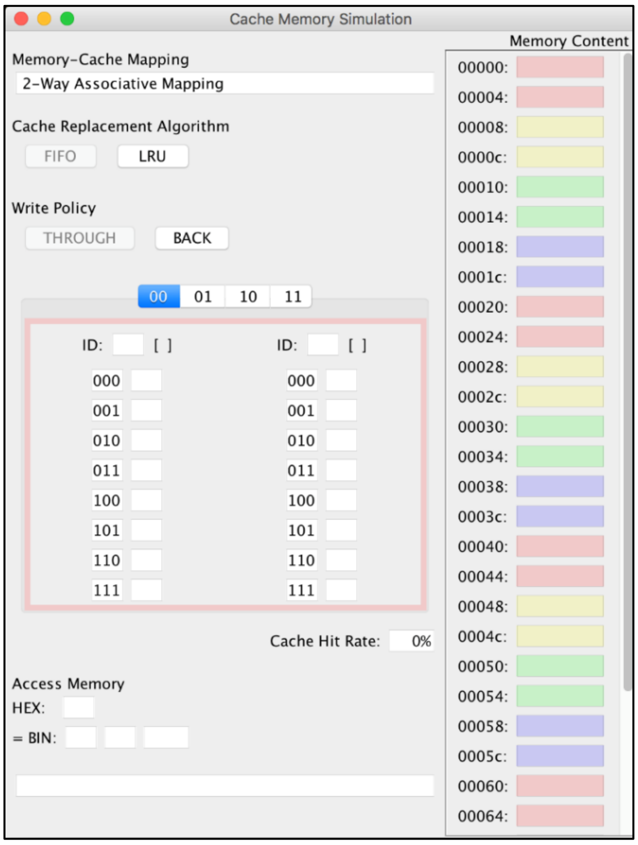
Cache Hit Rate: 0%

Figure 12.1 interface for cache memory simulation on Windows

The other major component of the final deliverable is cache memory simulation. With limited size in the cache, there are different schemes to determine how to access and replace the cache content. The simulation will allow the user to choose from two most commonly-used schemes covered in COMP2120, namely FIFO(First-In-First-Out) and LRU(Least-Recently-Used). For write policy, both write through and write back will be supported. As cache configuration, 2-way associative mapping is used with a total of four cache sets. These choices allow simple and easy graphical representation on one hand, and on the other, are some of the most commonly used options in real-life cache memory.

Figure 12.1 shows the completed interface of the cache simulation window on Windows, while figure 12.2 shows that on macOS. The major difference between the two interface lies in the difference of the color scheme. Different colors are used to signify different cache sets, however, during development, it is found that there are problems updating the memory content on Windows if the colors used have an alpha value less than 100, as in the implementation on macOS and Linux. Therefore the colors used for Windows environment are changed.

This cache simulation window will show in a separate window beside the panel of CPU simulation when the "Cache Simulation" button on the left in figure 10 is clicked. Note that the "Cache Simulation" button will be disabled when the simulation has begun, meaning that the cache simulation has to be initialized before then. The same applies to cache replacement algorithm and write policy, which have to be chosen by clicking the corresponding buttons before the simulation begins and cannot be altered during the process.


Cache Memory Simulation

Memory-Cache Mapping: 2-Way Associative Mapping

Cache Hit Rate: 0%

Figure 12.2 interface for cache memory simulation on macOS/Linux

Despite the fact that cache memory in real-life is always limited in size, but the small size of a few KB commonly in the context of real-life CPU execution is not trivial at all to be on a visual display. Therefore, the size of the simulated cache are limited to eight cache lines divided in four sets, which is also one analogy used in the course COMP2120. The cache content are displayed in a tabbed panel with four different tabs each graphically representing a cache set. Upon accessing the cache line in a cache set, the tab of will automatically be redirected to the corresponding cache set in action.

The cache hit rate textfield at the lower right of the cache tabbed panel dynamically updates the cache hit rate.

Under the text label "Access Memory" are several textfields labelled "HEX" and "BIN" the shows the memory address in concern and provide a clear indicator to how the cache replacement or memory write happen.

The "HEX" field contains two bits of hexadecimal address. Although as seen from figure 11 earlier, that the required address is of 5 bits length in hexadecimal representation, summing up to an addressing space of 1,048,576 bytes, this "enormously" sized addressing space is not possible to be displayed clearly on this simulator. Therefore there is a limitation to the addressing space of this simulator, which is 128 bytes, translating to a maximum of two bits hexadecimal address or seven bits binary address.

The "HEX" and "BIN" textfields are specifically designed to display the address information. The "BIN" textfields are divided into three parts. The leftmost one contains two bits specifying the block ID of the address concerned. The middle field contains two bits specifying the set number and the rightmost the remaining three bits offset in block.

When a memory block is first mapped to a cache line, i.e. when there is a cache miss, the textfields under "Access Memory" will automatically be filled with the corresponding data. The user can then trace back the block ID, set number of offset of the memory concerned with reference to the binary address conversion.

A point to note here is that the memory content on the right are represented in word-based slots, meaning that each slot contains a four-byte word and thus the memory address are all multiples of four. An implication to this point is that during programme input, the user cannot specify address that is not a multiple of four.

Since the offset is three binary bits, for each cache line representation, there are eight two-bit slots containing hexadecimal data which are segments from the word stored in the corresponding memory.

Lastly, there is a long rectangular textfield at the bottom of the cache simulation panel. This shows the description of cache miss or hit, and the block ID and set number involved.

The cache simulation is set into motion automatically when the “Simulate” button in figure 10 is clicked, given that the cache simulation window is prompted by clicking on the “Cache Simulation” button beforehand. When the simulation starts, the programme input entered by the user will first be loaded to the memory, updating the memory content of the cache simulation panel. As the simulation proceeds, the memory content will get updated dynamically when there is memory or write, as well as the cache set when there is cache access or replacement.

### Miscellaneous

This project is an educational project and the grand objective is to enhance teaching and learning experience. Hence, the sole delivery of the target simulator program may not be a very complete solution. To facilitate the use and understanding of this simulator delivered, sample definitions of twelve standard instructions will also be delivered, as attached in appendix 1. There will also be a user manual documenting everything that the user needs to know in order to operate the simulator.

# Methodology

CPU operations and memory caching in real life involves interactions of a lot of hardware and software components. To simulate the process, different models are constructed to have different states and data content simulating their real-life counterparts.

To better classify the different virtual objects created, different packages are defined to categorize different objects according to the purpose they serve. The two largest packages defined are the packages for the CPU simulation and the cache memory simulation, within which smaller packages are further created to better classify the different objects. The concept of package in Java programming can be simply viewed as directories containing the class files for better management and security measures.

## CPU Simulation

Figure 13 shows the structure of the classes developed for the instruction definition part of the CPU simulation. In this simplified diagram, different classes are modelling various components to be used in the simulation. There two main types of classes, one that is purely functional without any UI property, denoted by white boxes with dotted border and the other with UI properties denoted in yellow boxes. Functional classes stores the states and properties of objects that are used throughout the simulation, while UI classes are responsible for the visible UI components and are only used in some specific parts, without the connection other components.

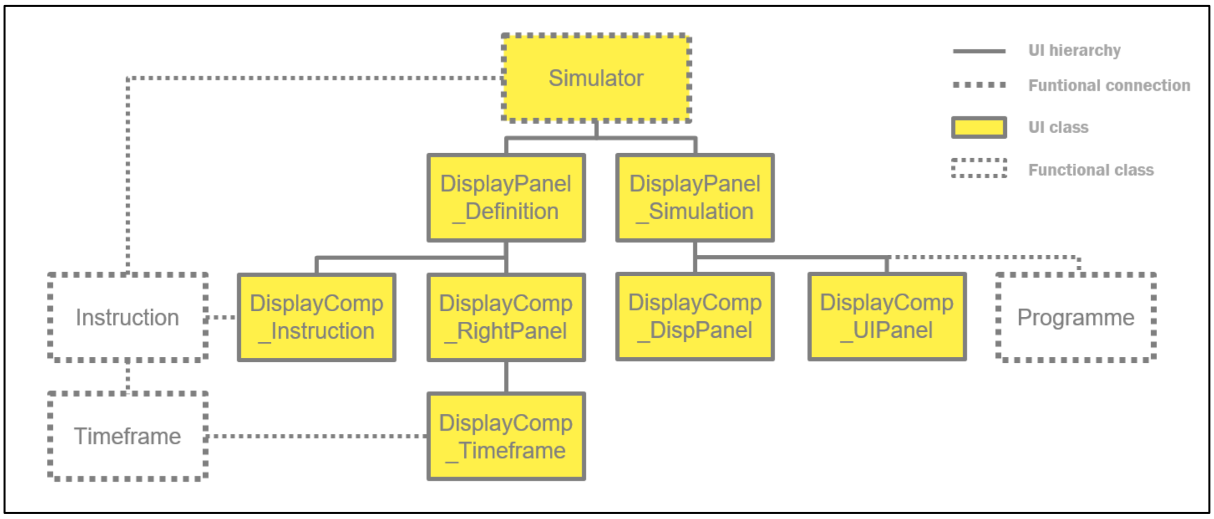


Figure 13. class structure of the CPU simulation

To provide a brief overview of how the different classes are defined and will interact, the Simulator class serves as the starting point. When the simulator program runs, the main function in the Simulator class will be invoked and a singleton Simulator object will be initialized. The Simulator object contains two different display panel corresponding to the two parts in the CPU simulation. Note that this “Simulator” class is the same class that dominates in the part of Cache Simulation which will be covered later, but for the sake of a better focus, the classes for cache simulation is not shown in figure 13 and will be discussed in the next subsection.

In the display panel of instruction definition, instruction objects will be created per instruction entry on the display, i.e. “DisplayComp_Instruction”. In other words, as graphical objects responsible for the user interface possess different behaviour from the functional objects that actually model the corresponding components, the rule of thumb for design of the class relationship is that, whenever necessary, for each functional objects that will be used throughout the simulation, e.g. “Instruction” modelling the actual mnemonic instruction defined, a corresponding graphical controller object will cater for its display properties, e.g. “DisplayComp_Instruction”.

The display panel of instruction definition is essentially a collection of the “DisplayComp_Instruction” objects, with the buttons and textfields that form the entire interface for the first part of the CPU simulation. Besides these objects, there is the other “DisplayComp_RightPanel” that serves as the division on the right as shown in figure 6, containing the array of “DisplayComp_Timeframe” that correlate with the “Timeframe” objects for modelling the sequence of execution timeframes of the instruction.

The remaining classes in figure 13, similarly, correspond to the second display panel of simulation display in the CPU simulation. There will be the "DisplayPanel_Simulation" which is the parent panel containing all the visualization materials and the programme I/O interface. More specifically, "DisplayComp_DispPanel" corresponds to the graphical display as seen on the left of figure 10. An image of the CPU architecture serves as the background, with data signal objects moving on top to simulate the operations inside the CPU. On the other hand, "DisplayComp_UIPanel" contains all the textual display of the content update of the CPU components and the programme input text area.

As the mentioned above, while the objects designated for graphical interface are basically independent across the different parts of the CPU simulation, as covered by highlight in yellow in figure 13, the functional objects, on the other hand, are connected to their respective interface classes and the class which utilizes their properties and behaviour, i.e. "Simulator", for sharing necessary data content, and are denoted by grey dotted border.

In order to differentiate the graphical and functional objects more easily and enhance the readability of the source code, the classes follow certain naming conventions. For classes designated for graphical interface, their names start with the word "Display". Depending on the scale of the objects to the entire simulation, the names are then followed by "Panel" for a more significant display role, or "Component" for a lesser scope. And ending with the words describing the functionalities. Functional objects, on the other hand, are not named by "Display" or "Panel", but rather by the direct naming of their functionalities or the objects being modelled.

One exception is the Simulator class, which possesses dual properties as a both graphical and functional object. The Simulator contains the main function at which the program is started and variables referencing to the functional objects defined; but it is also responsible for the display of the main simulator window. For the sake of simplicity, the direct naming of "Simulator" is used for this CPU simulation as well as for the cache memory simulation.

Another point to note is the "Programme" class belonging to the Simulation package, which is more of a redundant interface-like class now, rather than an object class. As the original design at the very beginning was to have an object class modelling the assembly programme which would be used throughout the simulation for referencing the programme input. But at later stage of the development it turned out that the programme input could well be captured and accessed by storing in a global array variable under the "Simulator" class. This enhances the overall coherence of the program and reduces the unnecessary inter-class references. However, since the methods of the "Programme" class have been constructed and extensively used by the time this update was realized, the class together with its methods remain to avoid massive changes to the entire program. Instead removing the entire class, the only change was to update the reference to the captured programme input to a global array under the "Simulator" class and abandon the use of the array under the "Programme" class.

The above has provided a brief overview of the implementation principles adopted in CPU Simulation. To understand better how the classes work together, an example worth discussing is saving the custom instruction definitions.

As mentioned in previous sections, custom definitions can be saved by clicking the “Save” button in figure 10. The saving function updates the instances of the “Instruction” class belonging to the “Simulator”, from the input captured in the respective “DispComp_Instruction”.

Since “DispComp_Instruction” possesses UI properties such as buttons and textfields that are only applicable when it is initialized as visible components on the instruction definition panel, it is not used as a functional class that spans throughout the entire simulation. Instead, instances of “Instruction” are created as variables under the “Simulator” class that can then be accessed whenever needed during the simulation. The “Save” captures useful, functional properties such as mnemonic symbol and opcode from the UI class to the functional instance.

The implementation principles of instruction definition has been generally discussed until this point. However, its development centered mainly around UI layout and interactions, which did not really touch down to the core logic of the simulation and will not be covered in any further detail in this report. Instead, the mechanism of running and displaying the simulation will be discussed as follows.

The execution logic of the simulation is written in the “DisplayComp_DispPanel” class. It is placed under the UI class due to the fact the animated simulation is achieved by repeatedly updating the visible UI, and drawing the graphics at slightly different locations at each update. Since the execution itself has no obvious functional counterpart, the logic is simply placed under the corresponding UI class to keep the interaction direct and simple.

The method to refresh the visible UI in Java is repaint(). To achieve the proper simulation animation, the question lies on how and when: how to control the locations of the graphics on the CPU background image at each frame, and when to call the repaint() method. Figure 14 shows a simple flow chart of the simulation logic.

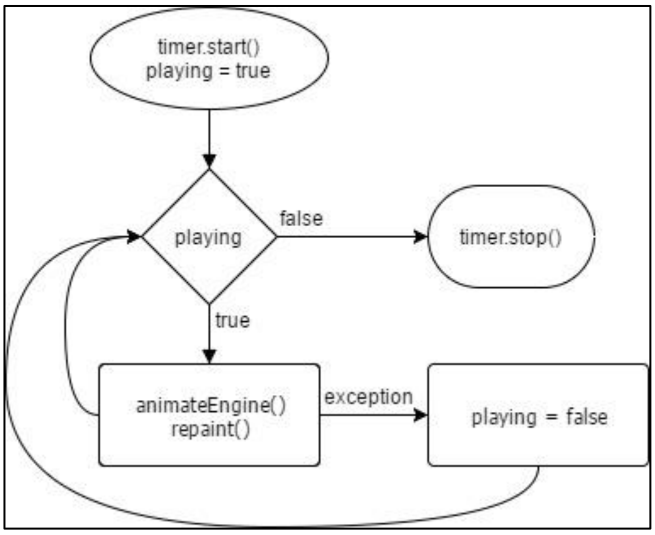
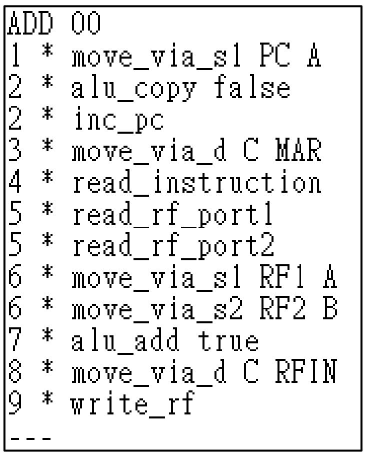


Figure 14. flowchart of simulation logic

To address to problem of how and when, the timer class from the Java.swing library is used. Basically, the timer object continuously executes the code provided to it for every given period of time specified by the caller. When the “Simulate” button in figure 10 is clicked and the simulation begins, the timer is started. The code supplied to the timer object is an if-loop wrapping a function called animateEngine(). The if-loop checks for a global variable in the “Simulator” class called “playing”, which is set to true when the simulation begins and false when the simulation halts or pauses. This if-loop in the timer object is executed every 20ms when the simulation begins - the default “speed” of the animation. By varying this period in which the timer repeats its execution, the speed of the simulation can be varied.



```
ADD 00
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_add true
8 * move_via_d C RFIN
9 * write_rf
---
```

Figure 15. sample definition of “ADD” instruction

The timer object indeed controls the “when” problem, for solution to the “how”, understanding the implementation logic of animateEngine() is needed.

Figure 15 shows the sample definition of the “ADD” instruction. To animate the instruction from the first timeframe to the last, visual representation of each and every of the timeframe has to be first determined.

Therefore, before figuring out how to implement the higher level animateEngine(), the problem is pushed deeper down to how to model every CPU action. Recall the elementary actions listed in table 1 which are the building blocks of all instructions supported by this simulator, with reference from the sim simulator[4], the graphical effects and actual functional consequences of simulating each of the actions were determined and are presented in here table 3. For better reference of the graphical effects, the CPU architecture image used in the simulator is attached in appendix 2.

| **Action** | **Simulation Effect** |
|---|---|
| move_via_s1 | update value of the data point object from the variable storing the value of the data source, graphically move the data point from the data source to the destination along the S1-Bus, update value of the destination with the value from the data point object |
| move_via_s2 | same as the above, but graphically along S2-Bus |
| move_via_d | same as the above, but graphically along D-Bus |
| inc_pc | move data point object graphically from PC to IncPC, then from IncPC to PC, increment the variable storing the value of PC by four when graphically reached PC in the end |
| read_rf_port1 | update value of the data point object from the variable storing the value of the required register, move data point object graphically from Register File to RFOUT1, update variable storing the value of RFOUT1 from the value of the data point object when graphically reached RFOUT1 |
| read_rf_port2 | same as the above, but with RFOUT2 |
| write_rf | update value of the data point object from the variable storing the value of RFIN, move data point object graphically from RFIN to Register File, update the value of the concerned register from the value of the data point object when graphically reached Register File |
| alu_add | update values of the data point objects from the variables storing the values of the temporary registers A and B, move the data points from A and B simultaneously to ALU, when graphically reached ALU, perform addition on the values of the data point objects, update the value of the first data point object and discard the other, move the |

[4] sim.cc. Hong Kong: HKU COMP2120

| | |
|---|---|
| | data point graphically to C, update the variable storing the value of C when graphically reached C |
| alu_sub | same as the above, but subtraction is performed |
| alu_and | same as the above, but bitwise-and is performed |
| alu_or | same as the above, but bitwise-or is performed |
| alu_not | same as the above, but with only temporary register A in action and bitwise-not is performed |
| alu_copy | same as the above, but with no ALU operation performed functionally, mere update of variables and moving data point graphically |
| read_instruction | update value of the data point object from the variable storing the value of MAR, graphically move the data point object from MAR to External Memory, perform a memory read based on the value of the variable of MAR, update the value of the data point object from the content fetched from the simulated memory or cache, move the data point object graphically from External Memory to IR, update the variable storing the value of IR from the value of the data point object, graphically move the data point object to Register File |
| read_memory | update value of the data point object from the variable storing the value of MAR, graphically move the data point object from MAR to External Memory, perform a memory read based on the value of the variable of MAR, update the value of the data point object from the content fetched from the simulated memory or cache, move the data point object graphically from External Memory to MBR, update the variable storing the value of MBR from the value of the data point object |
| write_memory | update values of the data point objects from the variables storing the values of MAR and MBR, graphically move the data point objects from MAR to External Memory and from MBR to External Memory, perform memory write based on both the values of the variable of MAR and that of MBR, update the variable storing the memory content depending on the cache policy |
| branch | check for branching condition, if branch, update the value of the data point object from the variable storing the value of MBR, move the data point object graphically from MBR to A along S1-Bus, update the variable storing the value of A from the value of the data point, move the data point graphically from A to ALU and to C, update the value of C from the value of the data point, move the data point graphically from C to PC along D-Bus, update the variable of PC from the value of the data point when reached PC, skip all remaining actions in the instruction and start the next instruction cycle |
| dec_dp | update value of the data point object from the variable of SP, move the data point graphically from SP to +-SP, decrement the value of the data point object by four, move the data point graphically from +-SP to SP, update the value of SP when reached graphically |
| inc_sp | same as the above, but increment by four instead |

| | |
|---|---|
| mar_to_temp | update the value of the data point object from the variable storing the value of MAR, move the data point graphically from MAR to TEMP, update the value of TEMP when reached graphically |
| temp_to_mar | update the value of the data point object from the variable storing the value of TEMP, move the data point graphically from TEMP to MAR, update the value of MAR when reached graphically |
| halt | basically no graphical representation on the simulation, except stopping the moving data points and updating the description fields to signal a halt |

Table 3. summary of the simulation effect of every hardware-level action

Breaking down the simulation to the action level is a step closer to realizing the implementation logic, however, there are still certain level of abstraction in the simulation effects described in table 3. For example, in the system bus related actions, what is the actual meaning of “move the data point from the data source to the destination”?

Taking a look at the CPU image in appendix 2, one can realize that it takes some diligence to display the animation of move via s1 from PC to A, as stated in the first action in figure 15. It requires moving the data point graphically from PC to the right, until touching the line of the S1-Bus, then moving upwards to the y coordinate at the line representing the circuit connecting to A, then moving to the left to enter A.

Recall the fact that animation is achieved by repeatedly calling repaint() after deciding what and where to paint in animateEngine() as shown in the flowchart in figure 14, this means the operations inside animateEngine() should not be too complex to avoid any lag in frame update, and should only update the graphics a little, e.g. a change of 2 or 3 pixels from the previous position, in order to realize a continuous visual effect. This suggests that the effects of the CPU actions, as listed in table 3, have to be further broken down, to a level of moving the graphics by several pixels.

A simplified diagram showing the implementation logic of animateEngine() is in figure 16.

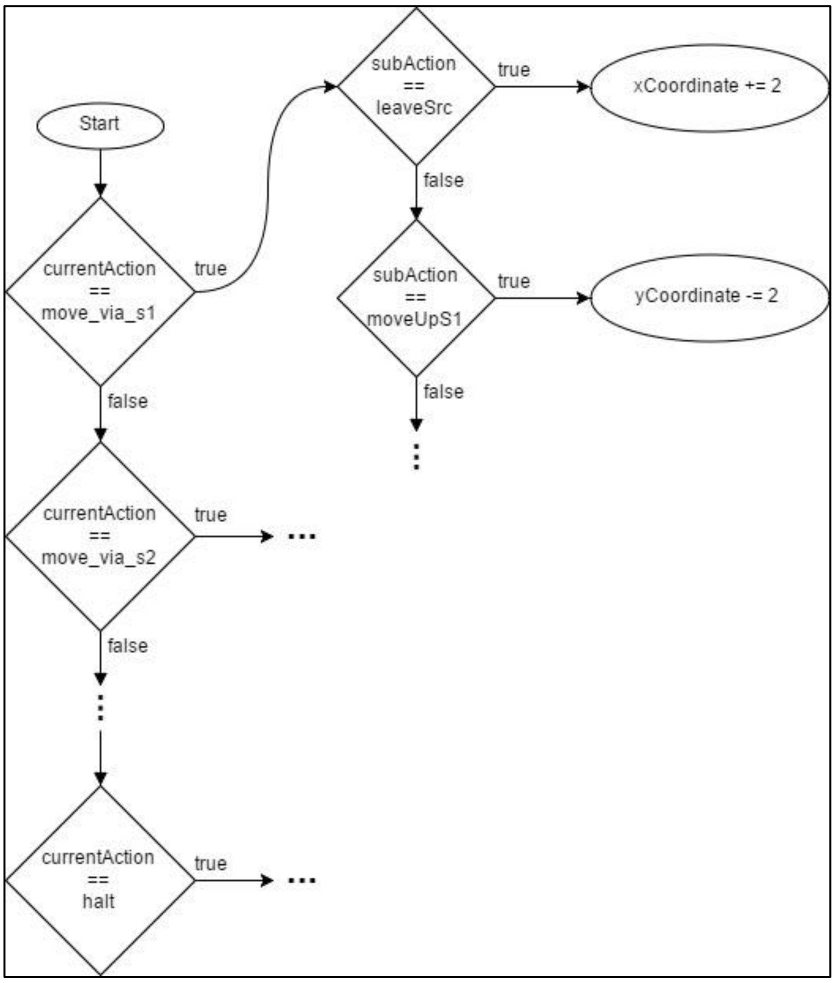


Figure 16. simplified flowchart of animateEngine()

animateEngine() is a method to update the coordinates of the data points for the repaint() method to draw the graphics later. It essentially is two or more levels of nested if-else statement.

When the simulation begins, the content at address “00000” is captured with the information of the instruction concerned, including the opcode, the operands and the sequence of CPU actions. A thread is then started to run the timer. Separation from the main execution thread is crucial as the loop of the timer to repaint the simulation display will hang the other UI components if being executed in the same thread. The timer will go through the actions defined in the instruction according to the index number, and repeatedly calling animateEngine() during the process. In animateEngine(), the current CPU action of the instruction that is being simulated will be checked against all the possible twenty-two alternatives to decide what to be performed. If there is a match - take the case of move_via_s1 in the definition of figure 15, the if-block of “move_via_s1” will then be entered. To goal is to reduce the execution to the lowest level of update by several pixels, therefore, inside the if-block of “move_via_s1”, the current status of the data point will be further checked against another set of if-else statements, called the subActions, which are conditions specific to that particular action. In the case of “move_via_s1”, the initial subAction will be set to “leaveSrc” that in the if-block, the data point will gradually move away from the source to the right by having its x coordinate increased by 2 pixel every time animateEngine() is called.

Challenges on animating the simulation with this approach is on one hand the complex construction of the nested if-else statements with tens of possibilities, on the other, it is the handling of parallel executions. As for every iteration in the timer(), updates to the graphics may be needed for multiple data points representation parallel CPU actions within a single timeframe instead of just handling a single point of execution, multiple instances of data may have to checked and updated according to the definition of the instruction, which adds to the complexity of the animation logic.

The solution multiple parallel execution is the extensive use of arraylist. Upon entering a new timeframe of simulation, a set of arraylists are initialized respectively for the current actions of the data points, the sub actions of the data points, the coordinates of the data points etc. Whenever there is an action entry in that timeframe, as captured from instruction definition, a corresponding item will be added to the respective arraylist. Then in the execution of the animateEngine(), every if-else for multiple times against all the entries in the arraylists. A single data point representing serial execution is only a special case in which there is only one item in all the arraylists.

## Cache Memory Simulation

The design of classes in the cache memory simulation follows the same paradigm used in CPU simulation, a simple class diagram is shown as figure 17.

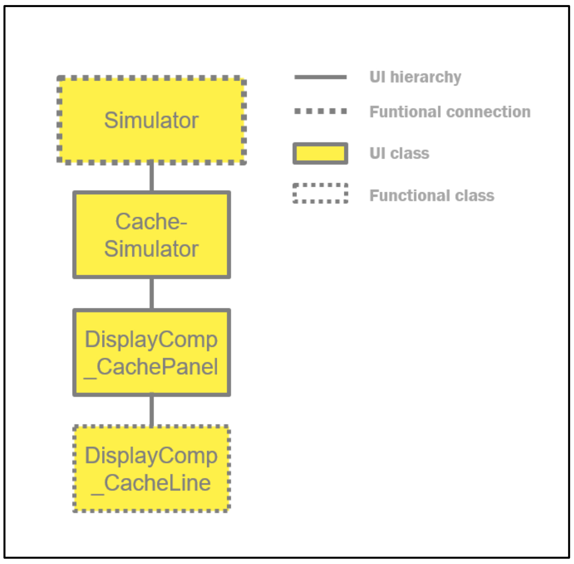


Figure 17. class structure of the cache memory simulation

Note that the class diagram for this part is much simpler than that of CPU Simulation. This is due to the fact that the simulation logic of decoding and running the programme input is already catered in the part of CPU Simulation, and Cache Memory Simulation requires almost no user input, but rather a simple one-way display from the program to the user, resulting in a smaller development scale.

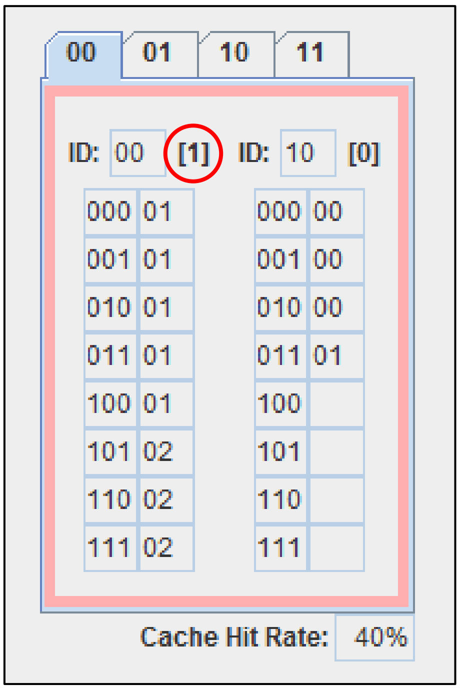
The classes utilized in Cache Memory Simulation interact with the same "Simulator" class seen in CPU Simulation, which serves as a global reference point for all simulation related data. Due to this approach, the design realized in figure 17 is different from the conceptual design mentioned in the previous report.

With this class structure, the number of classes are kept at a minimum by combining the functional and UI classes into single classes. The "CacheSimulator" class is responsible for the cache simulation window and the general layout. The "DisplayComp_CachePanel" is the individual panels for the cache sets in the tabbed panel see in figure 12.1 and figure 12.2. The "DisplayComp_CacheLine" controls the textfields of the cache line data within each set. No functional class is constructed separately for the cache lines because the cache lines themselves do not really contain a lot of specific properties, instead most of the properties such as the ID or data content, can be either accessed in other places or directly retrieved from the textfields. The same applies to the memory content, which are modelled as an array under the "Simulator" class and can be used to construct the UI representation as seen on the right of figure 12.1 and 12.2, without the need to explicitly construct another memory class.

The main focus of this subsection is on understanding the implementation logic of the cache replacement and memory write policy, while the details on UI development, similar to the previous section, will not be covered.

There are three places at which cache operations can be triggered. They are the hardware-level action "read_instruction", "read_memoy" and "write_memory", which involves accessing the simulated memory. When "read_instruction" or "read_memory" is executed and the data point has graphically reached the External Memory section on the CPU image, the actual read memory protocol of this simulator will take place.

The central idea of simulating cache replacement is to maintain a list of access order for each cache set to serve as the reference from choosing which cache line to be replaced next. The cache line to be replaced will be indicated by a counter. An example counter is circled in red in figure 18.



| ID: 00 | [1] | ID: 10 | [0] |
|---|---|---|---|
| 000 | 01 | 000 | 00 |
| 001 | 01 | 001 | 00 |
| 010 | 01 | 010 | 00 |
| 011 | 01 | 011 | 01 |
| 100 | 01 | 100 | |
| 101 | 02 | 101 | |
| 110 | 02 | 110 | |
| 111 | 02 | 111 | |

Cache Hit Rate: 40%

Figure 18. highlight of cache line counter

In the case when FIFO is chosen, the list acts like a queue. Whenever there is a cache line access, the line is pushed into the tail of the list, if not already in the cache. The cache line at the head of the list will always receive the counter "1", indicating that it is the next to be replaced. Upon replacement, the cache line at the head of the list is removed and the counter "1" migrates to the new head of the list. For the case of LRU, the mechanism is basically the same, that the head of the list will always receive the counter "1" and be replaced next. The difference lies on updating list upon access or replacement. In LRU, the list no longer serves as a simple queue. When a cache line is accessed in a cache set, the corresponding cache line will be moved to the tail of the list, and the cache line at the head of the list will have its counter updated to "1".

Note that the implementation stated above is actually kind of redundant or unnecessarily complicated for the current configuration where there are only two cache lines in a cache set. The point that is worth mentioning here is that this approach, same as most of the other approaches used in the development of this simulator, does not bind to a specific condition, but are scalable to adapting other configurations for possibilities of future development or upgrade. Most limits specified in this project, e.g. the number of cache lines and cache set, the addressing space, the bit count effectively used to identify the memory location etc. are declared as variables at initialization of the object class or at compile time that the implementations of the methods are mostly independent of the values stored in this variables. With this said, there exists the possibility to easily change the current implementation from 2-way associative to n-way associative.

For writing to memory, which is triggered when the action “write_memory” is executed, it is basically straight forward to understand and implement. The part that requires attention here is the time to update the simulated memory maintained in the “Simulator” class.

When the write policy is write through, update to the memory array is immediate when the write memory call finishes, and then the UI will update accordingly. But when the policy is write back, the update to the memory only happens when the cache line concerned are replaced from the cache set. Since for every write action, there is first an attempt to access the cache to see if the memory is in the cache or not, the same approach as in reading memory. Therefore, write back is catered by adding a check in the memory access method, to see if the cache line that is being replaced has been altered or not, if yes, the memory array will be updated.

One last minor implementation detail worth noting here is the update of the cache hit rate. The working principle is simple. There are two variables, one store the access count to memory, the other stores the number of cache hit. Whenever the memory access method is called, the access count is incremented by one, while if there is a cache hit, the count of cache hit is incremented by one. And the hit rate is taking the division of cache hit count by the access count.

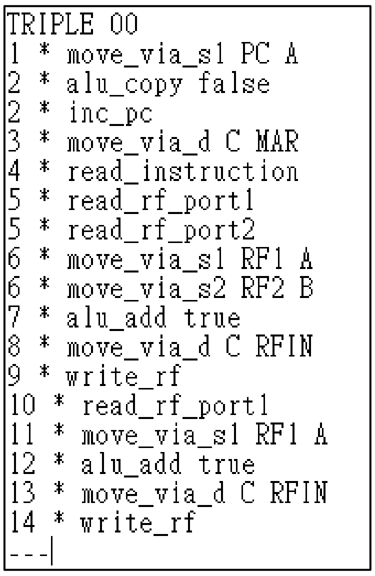
A general comment to this entire section of methodology is that some of the names of the classes and methods shown in the figures or mentioned, e.g. “DispComp_Instruction”, are not exactly as they appear in the Java code of the Simulator program, while the action strings listed in table 3 are all exact. These names are written this way in the report for the purpose of providing simple yet meaningful names in a clearer fashion, as some of the names used in the actual development have been declared since the beginning of the project and better refinements to the logical or structural design to the program have been made that these names in the code are not really precise in meaning but due to the extensive use, are kept untouched. Relationship between the names specified here and the actual classes or methods used in the code should be obvious to those who have read this section and the code.

## Experiments and Results

Technical specifications of the delivered simulator program and its functionalities have been described in the previous section of Scope and Deliverables. This section shall focus on the results and experience that users will get from using the simulator.

Comparing to the existing command-line simulator and that from the final year project last year, this simulator has achieved significant breakthrough in terms of the degree of practicability and flexibility of using this simulator. Users will have a very high degree of freedom to experiment with the simulator and gain practical knowledge about the lower-level CPU operations.

To demonstrate the possibility of this simulator, the following shows the sample definition of the “DOUBLE” instruction, an instruction that is probably not very useful and not seen in any existing instruction set.



```
TRIPLE 00
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_add true
8 * move_via_d C RFIN
9 * write_rf
10 * read_rf_port1
11 * move_via_s1 RF1 A
12 * alu_add true
13 * move_via_d C RFIN
14 * write_rf
---|
```

Figure 19. experimental "DOUBLE" instruction definition

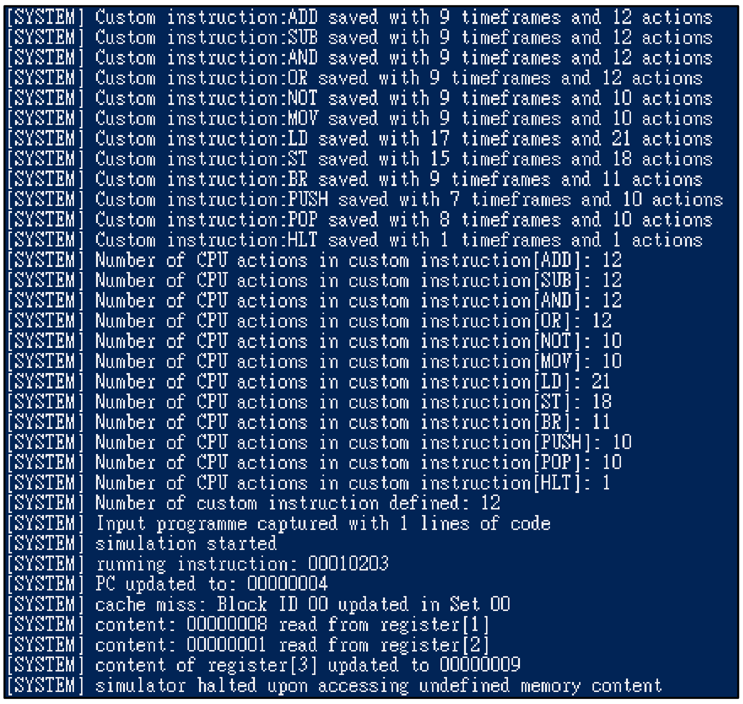
The "TRIPLE" instruction takes the values from the registers specified in operand 1 and operand 2, and triples the value when both registers specified point to the same register.

The functionality of this instruction is achievable by repeated "alu_add" in the definition, when the two registers supplied to the instruction point to the same register file, the value retrieve will then be added to itself twice, resulting in a tripled value. With standard addition instruction commonly seen in different instruction sets, the result will be a double when the two registers specified point to the same register file. It is because of the flexibility that more than one "alu_add" can be added in the definition that this instruction can render a triple value.

Another possible experiment achievable with this simulator is on addressing modes. Currently the simulator supports two addressing mode, with the identifying bit "f" signaling direct addressing and "e" signaling displacement addressing. There is, nonetheless, room for unsupported addressing modes, due to the fact that the implementation for addressing modes are entirely up to the user's discretion, i.e. the way they arrange the actions in an instruction and configure the identifying bit will determine the addressing mode used. As seen from the sample definition of the "LD" and "ST" instruction in appendix 2, the decision of which addressing mode to execute depends entirely on the action definitions. Therefore, with proper configuration, this simulator can support any two addressing mode at the same time, and not necessarily direct and displacement addressing.

Apart from experimenting with the simulator and building custom instructions, this tool greatly enhances the experience and facilitates learning by providing users with clear and comprehensive responses. Not only does the simulator provides description textfields for both CPU and Cache Memory Simulation that the user can take reference from, they can pause the simulation at any moment to carefully study the descriptions or the simulation graphics before going on.

To get some extra output from running the simulator, it is advised that users should run the Java executable from a command line tool to capture the log generated by the simulator. Figure 20 shows some sample log message of running a simulation.

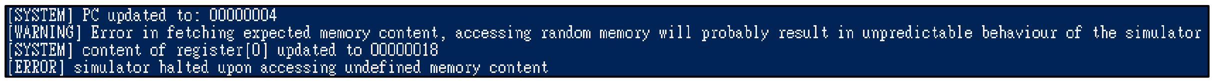


```
[SYSTEM] Custom instruction:ADD saved with 9 timeframes and 12 actions
[SYSTEM] Custom instruction:SUB saved with 9 timeframes and 12 actions
[SYSTEM] Custom instruction:AND saved with 9 timeframes and 12 actions
[SYSTEM] Custom instruction:OR saved with 9 timeframes and 12 actions
[SYSTEM] Custom instruction:NOT saved with 9 timeframes and 10 actions
[SYSTEM] Custom instruction:MOV saved with 9 timeframes and 10 actions
[SYSTEM] Custom instruction:LD saved with 17 timeframes and 21 actions
[SYSTEM] Custom instruction:ST saved with 15 timeframes and 18 actions
[SYSTEM] Custom instruction:BR saved with 9 timeframes and 11 actions
[SYSTEM] Custom instruction:PUSH saved with 7 timeframes and 10 actions
[SYSTEM] Custom instruction:POP saved with 8 timeframes and 10 actions
[SYSTEM] Custom instruction:HLT saved with 1 timeframes and 1 actions
[SYSTEM] Number of CPU actions in custom instruction[ADD]: 12
[SYSTEM] Number of CPU actions in custom instruction[SUB]: 12
[SYSTEM] Number of CPU actions in custom instruction[AND]: 12
[SYSTEM] Number of CPU actions in custom instruction[OR]: 12
[SYSTEM] Number of CPU actions in custom instruction[NOT]: 10
[SYSTEM] Number of CPU actions in custom instruction[MOV]: 10
[SYSTEM] Number of CPU actions in custom instruction[LD]: 21
[SYSTEM] Number of CPU actions in custom instruction[ST]: 18
[SYSTEM] Number of CPU actions in custom instruction[BR]: 11
[SYSTEM] Number of CPU actions in custom instruction[PUSH]: 10
[SYSTEM] Number of CPU actions in custom instruction[POP]: 10
[SYSTEM] Number of CPU actions in custom instruction[HLT]: 1
[SYSTEM] Number of custom instruction defined: 12
[SYSTEM] Input programme captured with 1 lines of code
[SYSTEM] simulation started
[SYSTEM] running instruction: 00010203
[SYSTEM] PC updated to: 00000004
[SYSTEM] cache miss: Block ID 00 updated in Set 00
[SYSTEM] content: 00000008 read from register[1]
[SYSTEM] content: 00000001 read from register[2]
[SYSTEM] content of register[3] updated to 00000009
[SYSTEM] simulator halted upon accessing undefined memory content
```

Figure 20. log generated from simulation

The simulator log messages that are helpful to understand the current status of the simulation, but may be too tedious to be displayed on the description fields on the simulator windows, such as the messages about the saved instructions as seen in figure 20.

There are three types of log message that the user may run into. The SYSTEM message that shows information about normal simulation status, for example current condition of the simulation or updates of values. The second type is WARNING message, which alerts users of undesired actions performed that may not have immediate consequence to the simulation but will lead to unpredictable results. An example of WARNING message is shown in figure 21.



```
[SYSTEM] PC updated to: 00000004
[WARNING] Error in fetching expected memory content, accessing random memory will probably result in unpredictable behaviour of the simulator
[SYSTEM] content of register[0] updated to 00000018
[ERROR] simulator halted upon accessing undefined memory content
```

Figure 21. example WARNING message

As this simulator has a finite addressing space, and for a lot of the memory slots, they will probably be empty, thus, there is a high chance that the user, without careful planning and implementation, will access memory address with no data content defined or beyond the addressing space limited. As in real-life situation, the memory contents other than those dedicated for the program are more or less random and uncontrollable to the current running program, accessing them may not lead to fatal error, especially for earlier machines with less error prevention measures. This WARNING approach follows the exact idea of not preventing the access of undefined memory content - probably accessed after mal-formed “push”, “pop” or “inc_pc” actions in the definitions, but returns random content and displays the WARNING message. However, despite that no immediate fatal error is discovered at the point of WARNING, this will most probably lead to undesired result, as seen from the last line in figure 21.

The last type of log message the user can get is the ERROR message, which indicates errors and exceptions in simulation runtime. ERROR messages suggest fatal errors and will immediately stop the simulation timer, causing the simulator to stop.

# Testing and Evaluations

Given the high flexibility of the simulator program, the possibility of users performing unexpected operations leading to errors is, unfortunately too, high. The ERROR message discussed in the previous section is a good example of catching the error and preventing the simulator to fail.

In order to minimize the chance of unexpected failure, precautions have been made.

## Error Prevention in Instruction Definition

Precautions are programmed with proper UI design whenever possible to restrict the possibility of error input from the first place. For example, the textfields for mnemonic symbol are capped at a maximum of six characters and the opcode field at two characters to ensure that the string can be properly handled and used in other parts of the simulation.

In other times, codes are written to validate certain situations. For instance, when the user saves the custom definition, a validate method will run to verify if the definition abides by some principal rules. Firstly, the mnemonic symbol and the hexadecimal opcode must be unique and cannot be duplicates from any of those already saved. Then, there will be a check for resource conflicts.

Since parallelization is a feature supported by this simulator, and during the execution of the CPU actions, different resources will be occupied, for example, if an “alu_add” is being executed, then any other ALU operations cannot be added in parallel within the same timeframe. The idea for checking is to specify certain resources involved in the particular action being used and locked, and post an error message if any parallel action utilizing the locked resources is found within the same timeframe. The list of resources exclusively used by each action that cannot be shared in parallel is shown in table 4.

| **CPU Action** | **Resource Locked** |
|---|---|
| move_via_s1 | data source, data destination, S1-Bus |
| move_via_s2 | data source, data destination, S2-Bus |
| move_via_d | data source, data destination, D-Bus |
| inc_pc | PC, IncPC |
| read_rf_port1 | RFOUT1 |
| read_rf_port2 | RFOUT2 |
| write_rf | RFIN, Register File |
| alu_add | A, B, C, ALU |
| alu_sub | A, B, C, ALU |
| alu_and | A, B, C, ALU |
| alu_or | A, B, C, ALU |
| alu_not | A, C, ALU |
| alu_copy | A, C, ALU |
| read_instruction | MAR, External Memory, IR, Register File |
| read_memory | MAR, External Memory, MBR |
| write_memory | MAR, External Memory, MBR |
| branch | - |

| | |
|---|---|
| dec_dp | SP, +-SP |
| inc_sp | SP, +-SP |
| mar_to_temp | MAR, TEMP |
| temp_to_mar | MAR, TEMP |
| halt | - |

Table 4. Resources exclusive used by actions that cannot be used in parallel

Other than that, mismatch or undefined resources will also be checked. If the user defines an instruction by manipulating the UI on the simulator window, i.e. selecting from drop-down boxes, there will be small chance of having this problem. However, the situation becomes unpredictable when the user imports a manually written text file containing inappropriate syntax or even a wrong file that does not abide by the syntax of the import/export text file at all. For example, by UI design, the available CPU actions are limited to the list of twenty-two action as in table 1, and the destination of “move_via_s1” available is limited to only the temporary register “A”. However, in an import text file, the user can put down an action string that does not belong to any of the twenty-two actions or can make the destination of “move_via_s1” to “PC”, which create situations that are impossible to simulate.

The solution to mal-formed or mismatched resources is that upon import of definition file, if there are incorrectly entered data as stated above, the resource concerned will automatically be updated to the first item in the drop-down boxes on the UI, e.g. any unrecognized CPU action will become “move_via_s1”. This will probably lead to unsuccessful simulation result if the user is unaware of the mistake, but at least this will not cause the simulator to fail or unable to respond and the user will notice the result of failed simulator after all.

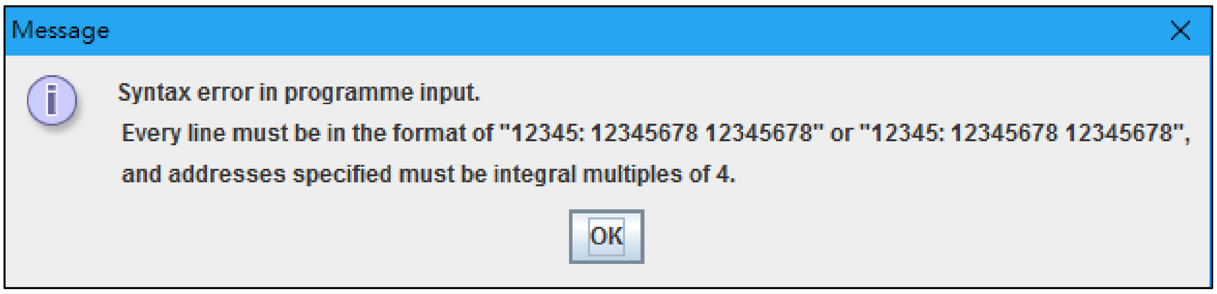
The following is a list of operations performed deliberately, concerning the part of instruction definition, to ensure the simulator can handle potential errors arose from them.

1. Add and remove timeframe entries in arbitrary numbers and order
2. Remove unsaved instructions in arbitrary orders and then save in arbitrary orders
3. Remove saved instructions in arbitrary orders and then export definition text file
4. Saved randomly only some of the instructions, then export definition file
5. Saved randomly only some of the instructions and then remove randomly some of the instructions, then export definition text file
6. Import a definition text file, then import another definition text file
7. Import a mal-formed definition text file, with mismatched resources
8. Import a mal-formed definition text file, with resource conflicts
9. Import a definition text file, then export definition text file
10. Import definition text file, remove arbitrary number of instructions and in arbitrary order, then export definition text file
11. Import a completely random text file which is not in the required format
12. Export definition file without saving any instruction

## Error Prevention in Simulation Display and Cache Memory Simulation

Possible errors from the simulation come probably from mal-formed programme input or internal fatal error during simulation caused by irrational programme logic, assuming that there is no mistake in instruction definition.

Preliminary precaution measures have been made to filter undesired syntax error from the input, an error prompt is presented in figure 22.



Syntax error in programme input.
Every line must be in the format of "12345: 12345678 12345678" or "12345: 12345678 12345678",
and addresses specified must be integral multiples of 4.

Figure 22. Error prompt on mal-formed programme input

As seen in figure 22, the simulator checks for the length of the code lines to ensure that the programme is properly formed with complete instruction words and the addresses are in multiple of four for proper indexing in the simulated memory array.

The rest that have been done to test for potential error during simulation is indeed, different test input for verification of expected result. The following figures shows the programme inputs that have been tested, each for a specific purpose. Note that the instruction set used for all these test programmes is attached in appendix 1.

```
00000: 01010101
00004: 0800ff00 00000000
0000c: 11000000
```
Figure 23. Test input for branch-always/goto

Figure 23 contains the code snippet that tests the branch-always execution. Note that the “branch” action depends on the condition code specified in operand 1, i.e. the 3rd and 4th bits of the instruction word. When the condition is “00” as seen in the figure, branching always occurs. The result of the above programme is a never-ending loop between the first and second line of code.

```
00000: 01010101
00004: 0801ff00 00000000
0000c: 11000000
```
Figure 24. Test input for branch-zero with zero-flag equals 1

Figure 24 contains the code that tests for branch-zero for zero-flag in value “1”. The condition code for branch is “01”, meaning that branching occurs when zero-flag is true. The first line of ALU operation sets the zero-flag to “1” and the result of the programme is again a never-ending loop between the first and second line of code.

```
00000: 00010201
00004: 0801ff00 00000000
0000c: 11000000
```
Figure 25. Test input for branch-zero with zero-flag equals 0

Figure 25 contains the code that is basically the same as that in figure 24 but with zero-flag set to "0" if the initial random value in register 1 and register 2 are not the same. The result of the programme is revoked branching and the simulation continues to line three and halted.

```
00000: 00010201
00004: 0802ff00 00000000
0000c: 11000000
```

Figure 26. Test input for branch-nonzero with zero-flag equals 0

```
00000: 00010101
00004: 0802ff00 00000000
0000c: 11000000
```

Figure 27. Test input for branch-nonzero with zero-flag equals 1

Figure 26 and 27 contain the inputs for the last variation of branching, the branch-nonzero execution, where the condition code is "02" meaning that the branch will occur if zero-flag is set to "0", as in figure 26, and revoked if zero-flag is "1", as in figure 27. The result of running the code in figure 26 is a never-ending loop between the first and second line of code, while that of figure 27 continues to halt.

```
00000: 01010101
00004: 01020202
00008: 0600ff01 00000020
00010: 0600e103 00000024
00018: 11000000

00020: 00000004
00024: 00000024
00028: 00000ffe
```

Figure 28. Test input for different addressing mode

Figure 28 contains the code for testing different addressing mode. The addressing mode identifying bit is specified in the first bit of operand 2, i.e. the 5th bit of the instruction word, hence effectively operand 2 for specifying the register file takes into account only the second bit, i.e. the 6th bit of the instruction word. This suggests that the maximum number of simulated internal register supported by this simulator is one bit of hexadecimal digit, i.e. sixteen.

At the 3rd line of the sample programme in figure 28, direct addressing is adopted and the data content in "00020" is loaded to register 1. At the 4th, displacement addressing is adopted, adding up the content of the second word and that stored in register 1, i.e. 24 + 4 = 28, to get the address for the "LD" instruction. The result of the programme is register 3 having the data content "00000ffe" in the end.

The following test inputs are dedicated for the testing of features implemented in Cache Memory Simulation.

```
00000: 01010101
00004: 01020202
00008: 0600ff01 00000040
00010: 0600ff02 00000000
00018: 0600ff03 00000060
00020: 11000000

00040: 00000001
00060: 00000002
```

Figure 29. Test input for FIFO and LRU cache replacement

Figure 29 shows the input for testing the difference in using FIFO and LRU. When running the code in the simulator with the different color scheme in the cache simulation window, it will be obviously that addresses “00000”, “00040” and “00060” used in the code are of the same set number and will be accessed via the same cache set. Due to 2-way associative mapping but three cache lines in concern, replacement is going to happen. In the case when FIFO is used, since “00000” is the first of the three to be referenced and read into cache, it will be replaced at line 5 when “00060” is read into cache. On the other hand, when LRU is used, since “00000” is accessed again in line 4, the least recently used cache line thus becomes “00040”, which will be replaced at line 5.

```
00000: 0701ff00 00000030
00008: 0600ff01 00000050
00010: 0600ff02 00000000
00014: 11000000

00050: 00000001
```

Figure 30. Test input for write through and write back

Figure 30 contains the code that tests write policy. Again, it is clear with the difference in color scheme provided by the simulator that “00010”, “00030” and “00050” belongs to the same cache set. When write through is used, memory content at “00030” will be directly updated at the first line of “ST” instruction. In the case of write back, since after the first line, “00030” is still in the cache without getting replaced, update to the memory content will not occur until the third line when content in “00010” replaces cache line of “00030”.

```
00000: 0701ff00 00000030
00008: 0600ff01 00000014
00010: 0600ff02 00000050

00014: 00000010
00050: 00000001
```

Figure 31. Failed test input for write policy

Figure 31 is a failed attempt to test for the difference between the two write policy. Despite the failure to fulfill its purpose, the code snippet is still recorded for it is worth investigating the cause of failure.

This code in figure 31 is a completely legitimate programme to be simulated. However, the expected write back behavior did not happen during testing. This was due to the fact that the third address competing for the cache replacement, i.e. “00050” was never accessed in the above code. The mistake is highlighted in red. Note that the red “00000050” on the third line is implicitly having the address of “00014” as it is the second word following the word at “00010”. Therefore the data word in the later line highlighted in red actually introduces new content to the memory slot and masks the content of “00050” defined previously.

This is quite an interesting mistake worth noting in this report and nothing has been done to prevent this from happening. This shall not be seen as a defect to the simulator program, but rather, a test of carefulness to the user when inputting their programmes for simulation.

The above test inputs are only some of the tests recorded and are displayed here because they involve the use of some rather tricky execution logic, e.g. the branching instructions and the different addressing modes, cache replacement and write policy etc. For sample instruction and actions that are more straight-forward, such as the ALU related instructions, tests have been carried out to ensure their functionality meeting expectation but are not recorded here.

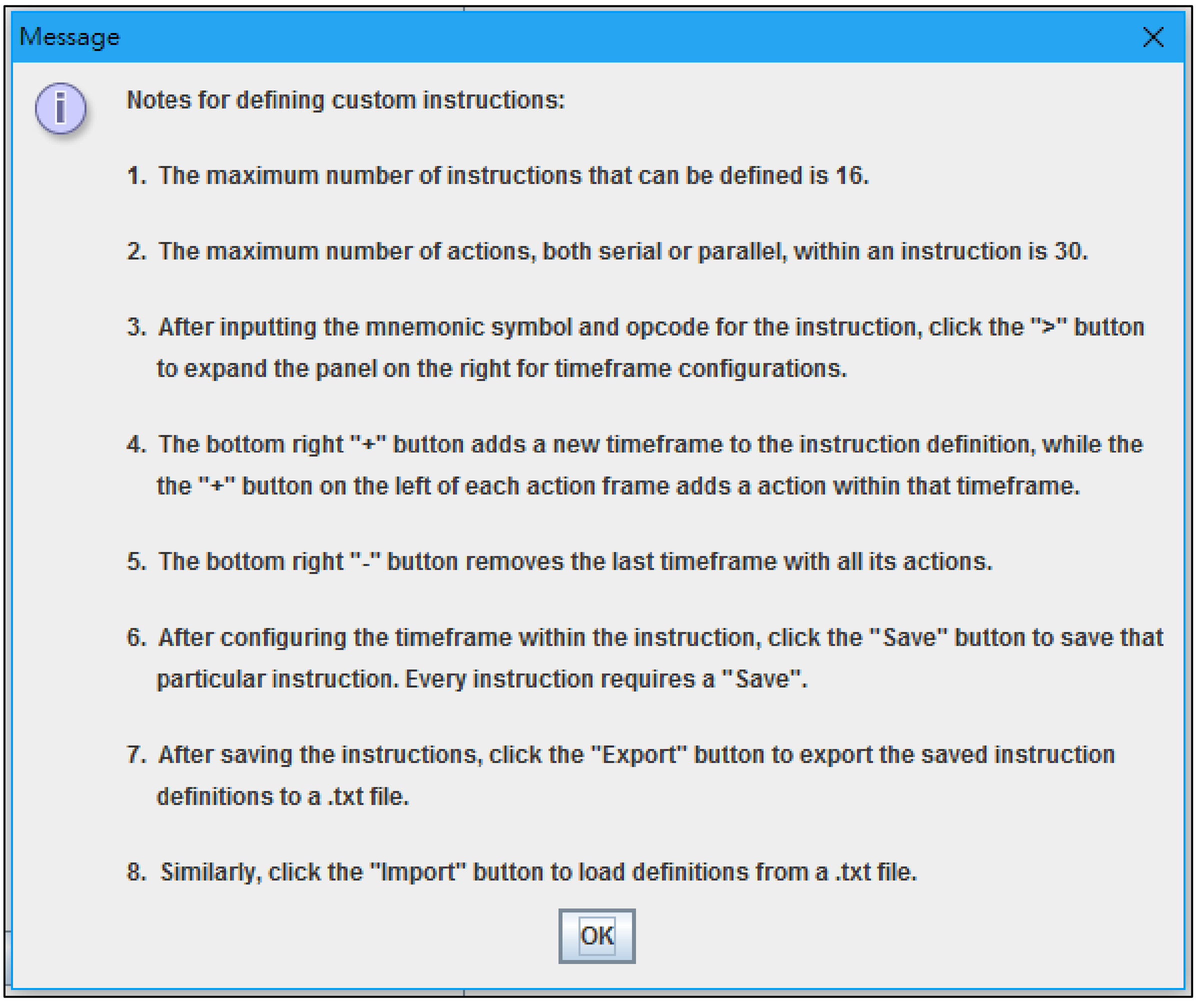
## Miscellaneous

This project is after all an educational project and one of the keys is to help users properly utilize this simulator. Comparing to the existing command-line simulator and the simulator from the final project last year, this simulator is indeed more intuitive to operate.

Despite the comprehensiveness and flexibility in functionality that this simulator provides, it is more intuitive in a sense that this is a standalone Java program with well-rounded UI and requires no external configuration file to operate. For the other two simulators, external configuration files are needed to couple with the simulator programs and the command to start the simulator or the content and format required by the configuration files have to be made clear to the user beforehand.

With this simulator, users can really kick start with no prior knowledge and experiment with the UI to find the proper way to do some basic simulations. Certainly, the help from a user manual or some operation principles about this simulator can consolidate the user’s knowledge to some implicit features, allowing them utilize this tool to the fullest.

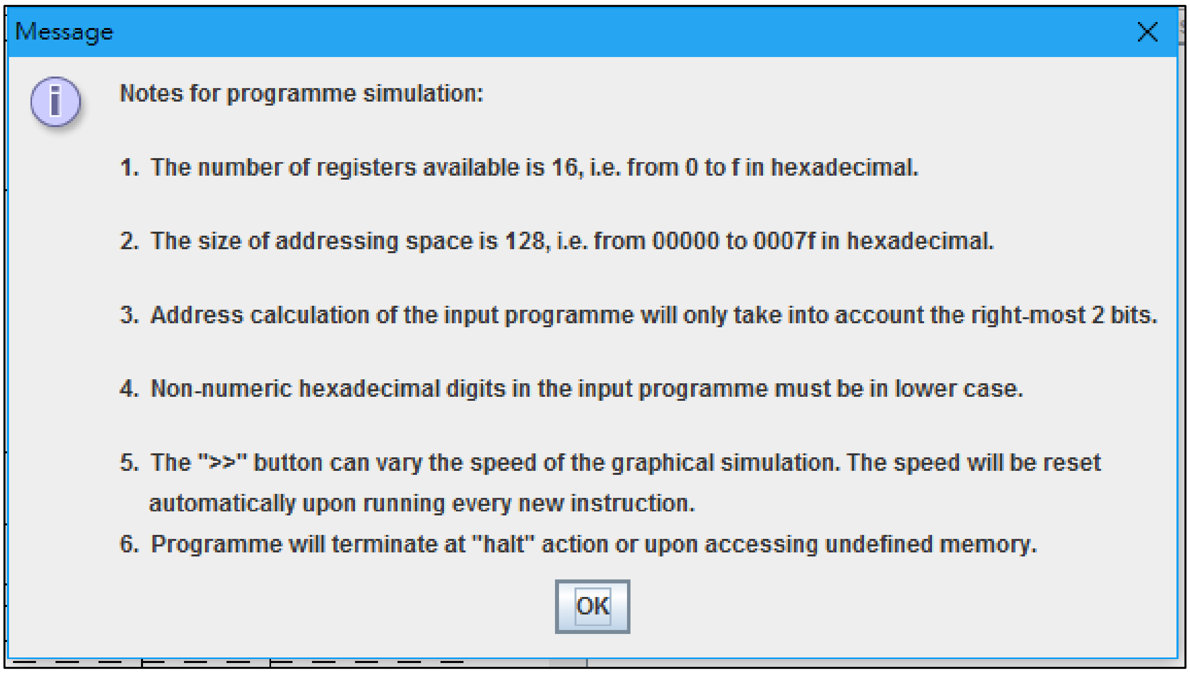
To better facilitate users to get the information about operating this simulator, besides the detailed user manual which will be provided, there are two immediate help prompts built in the simulator to offer some quick tips. The help prompts can be instantiated by clicking the “Help” buttons in shown in figure 6 and figure 10 respectively. Figure 32 and figure 33 shows the prompt windows the help messages.



Notes for defining custom instructions:

1. The maximum number of instructions that can be defined is 16.
2. The maximum number of actions, both serial or parallel, within an instruction is 30.
3. After inputting the mnemonic symbol and opcode for the instruction, click the ">" button to expand the panel on the right for timeframe configurations.
4. The bottom right "+" button adds a new timeframe to the instruction definition, while the the "+" button on the left of each action frame adds a action within that timeframe.
5. The bottom right "-" button removes the last timeframe with all its actions.
6. After configuring the timeframe within the instruction, click the "Save" button to save that particular instruction. Every instruction requires a "Save".
7. After saving the instructions, click the "Export" button to export the saved instruction definitions to a .txt file.
8. Similarly, click the "Import" button to load definitions from a .txt file.

Figure 32. Help prompt from the instruction definition window



Notes for programme simulation:

1. The number of registers available is 16, i.e. from 0 to f in hexadecimal.
2. The size of addressing space is 128, i.e. from 00000 to 0007f in hexadecimal.
3. Address calculation of the input programme will only take into account the right-most 2 bits.
4. Non-numeric hexadecimal digits in the input programme must be in lower case.
5. The ">>" button can vary the speed of the graphical simulation. The speed will be reset automatically upon running every new instruction.
6. Programme will terminate at "halt" action or upon accessing undefined memory.

Figure 33. Help prompt from the simulation display window

# Conclusion and Future Works

This simulator has opened up opportunities for users to exploit like never before. It offers the same level of comprehensiveness as the previous simulators for features that have long existed in the simulators: fetch and decode the programme input, provide comprehensible descriptive feedbacks on simulation status and the like. And it adds an novel feature of utmost importance to the list: the construction of custom instructions from ground zero. Yet on the other hand, this simulator is made more intuitive to use and less rigid to operate.

With more extensive features and higher flexibilities in usage, comes the cost. Unexpected results or even errors are more likely to prompt with this simulator, despite that huge effort has been invested in testing the program and preventing exceptions from happening. During the preparation of this report, test cases that have not been simulated before have been discovered, which unfortunately, may lead to failure of the program.

The following lists the possible scenarios for failure which have been discovered but not tested or catered until this point.

1. Discontinuous index numbers of actions in definition file imported may cause unexpected simulation behaviour
2. Parallel action with “branch” action may or may not be executed

The future works of this project will be on handling the above mentioned uncertainties and possibly evaluate for any other potential errors. There is no guarantee that this simulator program can be perfected to a level such that no error or failure will ever occur, rarely can any software of complexity in this world guarantee this. But the key lies in the attitude of learning and recovering from mistakes, there is always a way of closing the simulator program and starting an error-free simulation again.

With the above said, it is believed that unattended errors or failures are not common in this simulator program. While the searching for the possibility to future enhance the program, the other major focus will be on generating the detailed user manual. After all, this is an educational project and to enable users to utilize this tool so that they can enhance their learning and teaching experience, is of utmost importance.

To conclude, this project has delivered a graphical simulator that has hopefully met all the objectives and tasks that it has been intended for. While there is still a certain level of uncertainty in the program due to its dynamic nature, it can no doubt offer a comprehensive and exciting tool for learning and teaching in COMP2120.

## References

1. sim.cc. Hong Kong: HKU COMP2120.
2. Wong, Jing Hing(Kent). FINAL REPORT TOPIC: COMPUTER SYSTEM SIMULATOR. Hong Kong: 2015
3. Wong, Jing Hin. “FYP: Computer System Simulator”. i.cs.hku.hk. 2016. Thu. 1 Dec. 2016.

# Appendix 1

## Sample Definition of 12 Standard Instructions

```
ADD 00
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_add true
8 * move_via_d C RFIN
9 * write_rf
---

SUB 01
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_sub true
8 * move_via_d C RFIN
9 * write_rf
---

AND 02
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_and true
8 * move_via_d C RFIN
9 * write_rf
---
```

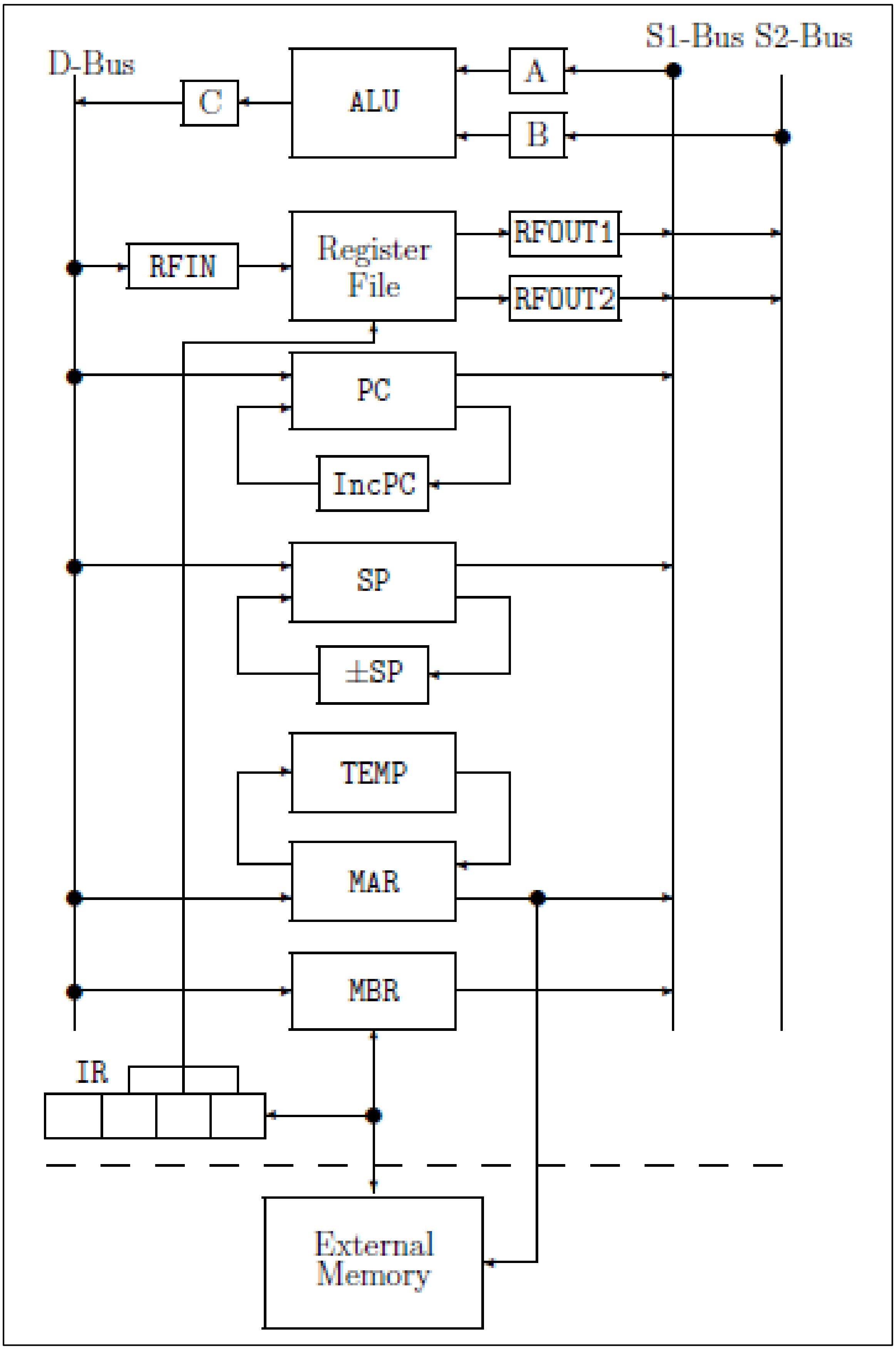
```
OR 03
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
5 * read_rf_port2
6 * move_via_s1 RF1 A
6 * move_via_s2 RF2 B
7 * alu_or true
8 * move_via_d C RFIN
9 * write_rf
---

NOT 04
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
6 * move_via_s1 RF1 A
7 * alu_not true
8 * move_via_d C RFIN
9 * write_rf
---

MOV 05
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
5 * read_rf_port1
6 * move_via_s1 RF1 A
7 * alu_copy true
8 * move_via_d C RFIN
9 * write_rf
---

LD 06
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
```

```
4 * read_instruction
4 * move_via_s1 PC A
5 * alu_copy false
5 * inc_pc
6 * move_via_d C MAR
7 * read_memory
8 * move_via_s1 MBR A
9 f alu_copy false
9 e read_rf_port2
10 e move_via_s2 RF2 B
11 e alu_add false
12 * move_via_d C MAR
13 * read_memory
14 * move_via_s1 MBR A
15 * alu_copy false
16 * move_via_d C RFIN
17 * write_rf
---

ST 07
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
3 * move_via_d C MAR
4 * read_instruction
4 * move_via_s1 PC A
5 * alu_copy false
5 * inc_pc
6 * move_via_d C MAR
7 * read_memory
8 * move_via_s1 MBR A
9 f alu_copy false
9 e read_rf_port2
10 e move_via_s2 RF2 B
11 e alu_add false
12 * move_via_d C MAR
13 * read_rf_port1
14 * move_via_s1 RF1 A
15 * alu_copy false
16 * move_via_d C MBR
17 * write_memory
---

BR 08
1 * move_via_s1 PC A
2 * alu_copy false
2 * inc_pc
```

```
3 * move_via_d C MAR
4 * read_instruction
5 * move_via_s1 PC A
6 * alu_copy false
6 * inc_pc
7 * move_via_d C MAR
8 * read_memory
9 * branch
---

PUSH 09
1 * dec_sp
1 * inc_pc
2 * move_via_s1 SP A
2 * read_rf_port1
3 * alu_copy false
3 * move_via_s1 RF1 A
4 * move_via_d C MAR
4 * alu_copy false
5 * move_via_d C MBR
6 * write_memory
---

POP 10
1 * move_via_s1 SP A
1 * inc_pc
2 * alu_copy false
2 * inc_sp
3 * move_via_d C MAR
4 * read_memory
5 * move_via_s1 MBR A
6 * alu_copy false
7 * move_via_d C RFIN
8 * write_rf
---

HLT 11
1 * halt
---
```

# Appendix 2

## Background CPU Image in Simulator


D-Bus
S1-Bus S2-Bus
C
ALU
A
B
RFIN
Register File
RFOUT1
RFOUT2
PC
IncPC
SP
±SP
TEMP
MAR
MBR
IR
External Memory