The University of Hong Kong

COMP4801 Final Year Project

Interim Report

Simulation of Simple Computer System Teaching

Wong Yin Lok
3035094933

20 Jan 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 aims at delivering two simulation components in one simulator program, one for CPU simulation and the other for cache memory simulation. Until the point when this report is written, development of the CPU simulation has completed by half and the remaining part is expected to be finished in the coming month, while development of the cache memory simulation is anticipated afterwards.

Acknowledgement

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

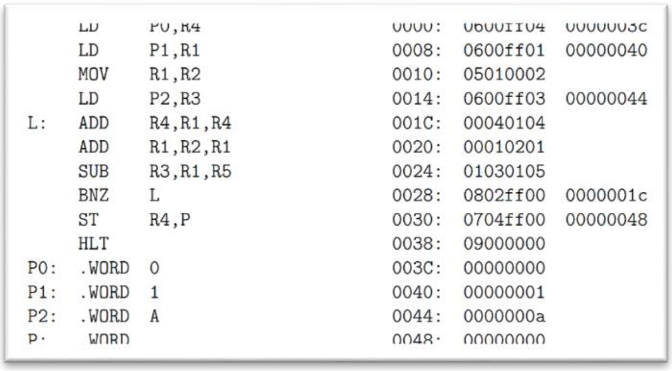
Table of Content

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 to indicate specific operations and data location according to a scheme referred to as the instruction set. Figure 1 shows an example program 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. simulation 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 pseudo code is written in a form like "ADD R4,R1,R4" while the actual code perceived by the CPU is like series of hexadecimal numbers corresponding to different operations and registers or memory address. Simply understanding the pseudo code but not being able to translate them into hexadecimal operation codes is not enough. Students do need to understand the translation.

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.

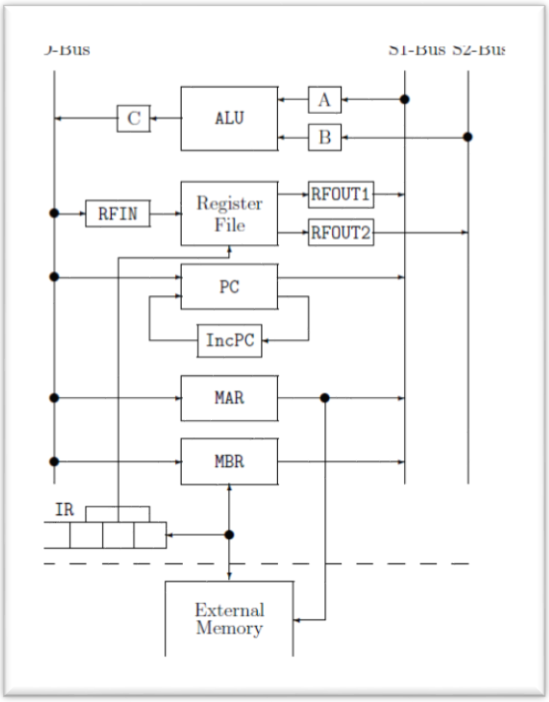
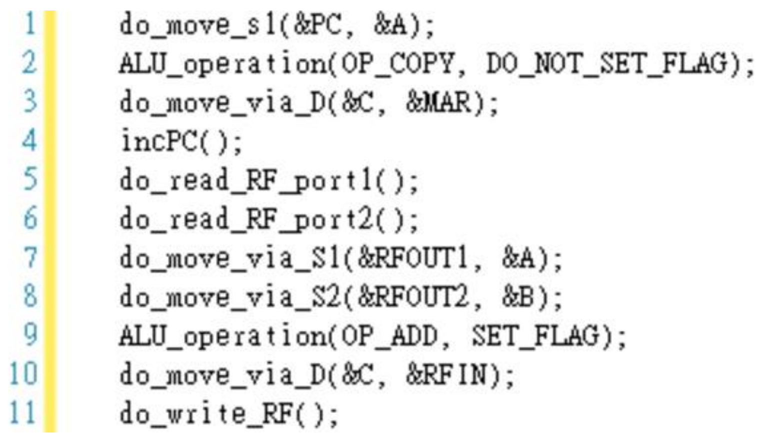


Figure 2. CPU architecture used in COMP2120

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 CPU implementations required for that single line of code.



```
do_move_s1(&PC, &A);
ALU_operation(OP_COPY, DO_NOT_SET_FLAG);
do_move_via_D(&C, &MAR);
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

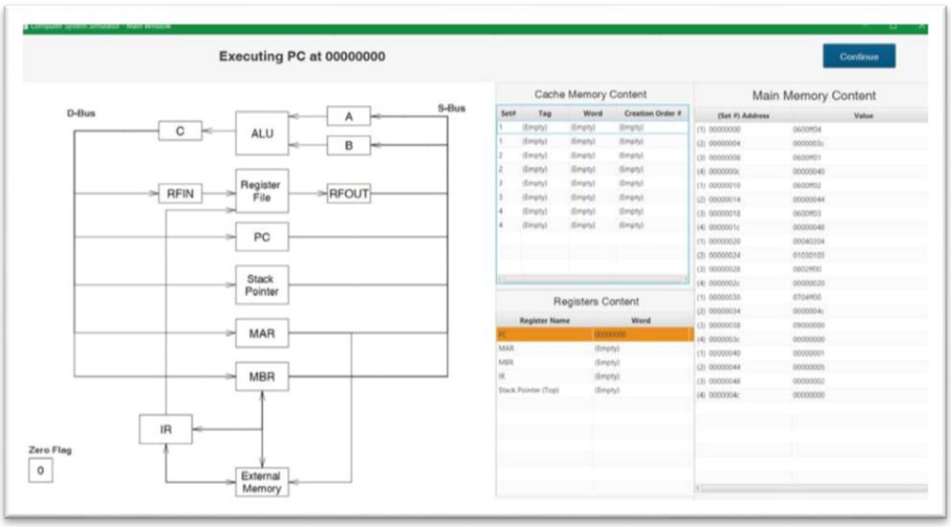
In addition, memory cache replacement can be a confusing process in which different slots of the memory cache are swapped out and replaced by new memory content. Memory caching is the process of reading comparably slow memory into fast-access CPU cache. While the number of slots and the size of the memory 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 3 shows a screenshot of the simulator from the project last year.



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

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.

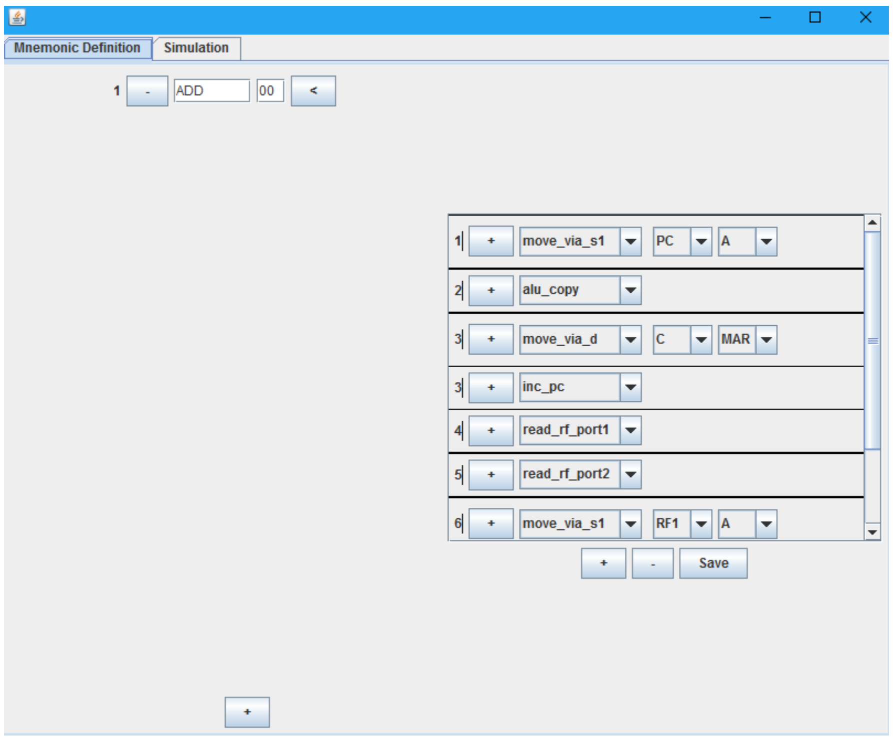
The key is to introduce the element of practical implementation, so as to install knowledge via solid practice and hands-on experience. This current project, therefore, basically starts all over again from scratch regarding the part on CPU simulation. For cache memory simulation, despite the fact that it was properly catered in the project last year, its development in this current project is partially dependent on the CPU simulation where the source codes of the two simulations interconnect. Thus, while the code on cache memory simulation from last year will be recycled where possible, the portion reused is expected to be trivial.

Scope and Deliverables

The final deliverable will be a Java program including two simulator components, namely the CPU simulation and the cache memory simulation. Java is chosen because of the reusability of previous code and the platform independent nature of the compiled program that 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.

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 5. As stated in previous sections, the objective of the CPU simulation is to allow users to learn and better understand the actual operations inside the CPU, via practical hands-on experience rather than merely watching the graphical display run. Users will be able to define the implementation of any single operation.



1 ADD 00

1 move_via_s1 PC A
2 alu_copy
3 move_via_d C MAR
3 inc_pc
4 read_rf_port1
5 read_rf_port2
6 move_via_s1 RF1 A

Figure 5. interface for mnemonic mapping of the CPU simulation

As seen from the above, 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 possibility for errors, the number of instructions that can be defined is capped at a maximum of 13, which should be more than enough for writing general programmes.

On the right, corresponding CPU actions that implement the desired function can be added and selected in the drop-down boxes. The index numbers in the CPU operation timeframe and the multiple add buttons here are worth noting. These are the design for parallelization. As different CPU operations involve 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. For example, when fetching an instruction, the program counter can be increased alongside the operations to copy the instruction to the instruction register without affecting the logic. As shown in figure 5, instead of initializing a new timeframe from index 3 to index 4, the subsequent CPU operation is added within the same timeframe such that the two operations can go in parallel denoted by the same index number.

Similar to the limit in instructions, there is a limit of 30 CPU operation timeframes that can be added for each instruction, regardless of the index number. This number should be more than enough for definition of common instructions.

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 6 shows the preliminary design of this part which is still under development.

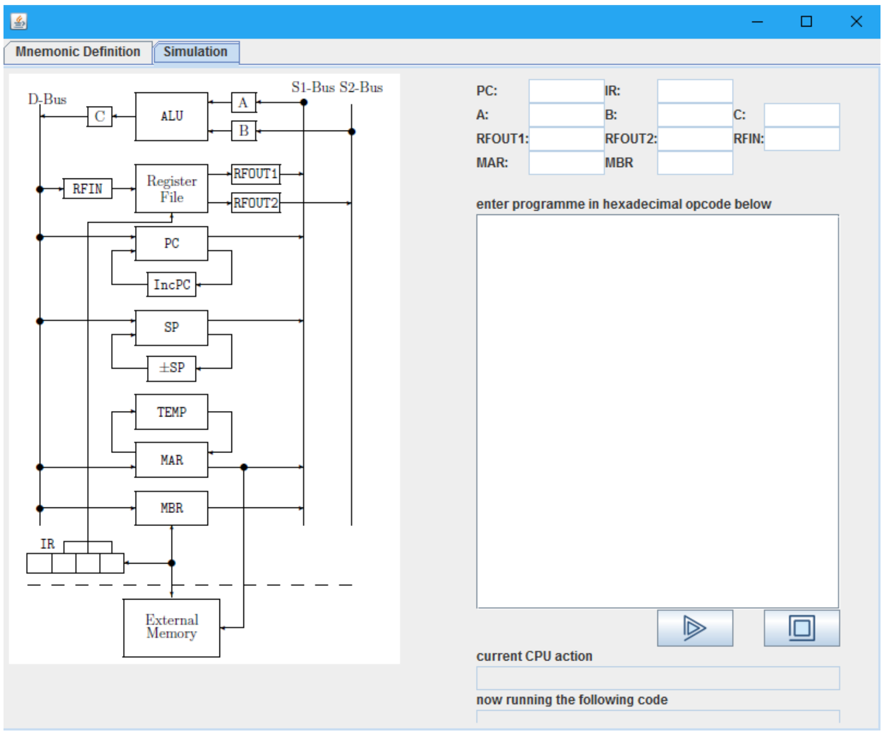


Figure 6. interface for graphic display tab of the CPU simulation

The simulator provides a large text field for users to type in their programme in hexademical operation 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.

When the play button is hit, there will be colored points, representing data signals, moving on the CPU image to indicate data movement. Relevant CPU modules on the path, e.g. MAR and PC, will have content update which will be shown in other description fields on the right beside the display.

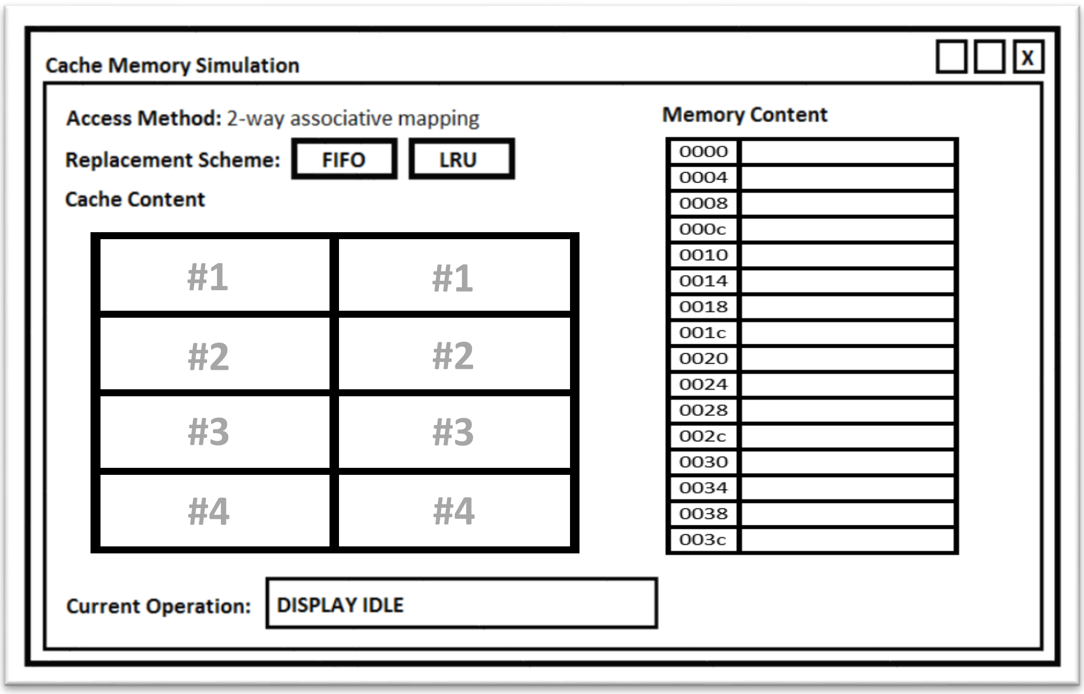
As students are required to programme with custom instruction sets in the course, this part of the simulation allows them to test their very own implementations and enhance understanding of the principle. However, it is not possible for the simulation to support definition of any unseen operations, because for operations that require the use of ALU – Arithmetic Logic Unit – to perform simple arithmetic like addition and subtraction, the ALU operations will have to be pre-defined by the simulation system. Therefore, this simulation will only guarantee the definitions of the basic operations listed in table 1, and the corresponding sample definitions will be delivered along with the final simulator.

Supported Operations
ADD
SUB
AND
OR
NOT
MOV
LD
ST
BR

Table 1. Operations supported by the simulator

The other major component of the final deliverable is the cache memory simulation. With limited size in the cache, there are different schemes to determine how to read in and replace the cache content. The simulation will allow the user to choose from different schemes. The implementation of these logics forms the basis of memory caching simulation. There are indeed a variety of practical replacement schemes, some of which are not covered in the course of COMP2120. As the key is to get students familiar with the general concept of memory caching, the memory simulator will be limited to the universal access method of 2-way associative mapping and the FIFO(First-In-First-Out) or LRU(Least-Recently-Used) replacement scheme.

Figure 7 shows the design of the memory simulation panel, which basically resembles that in the simulator last year. This panel of cache memory simulation will show in another window besides the panel of CPU simulation at the time when the simulator program runs. Users can choose between the cache replacement scheme by clicking on the corresponding buttons. By default, the replacement scheme is set to FIFO.


Access Method: 2-way associative mapping

Current Operation: DISPLAY IDLE

Figure 7. conceptual interface for cache memory simulation

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 actual CPU operations is not trivial at all to be on a visual display. Therefore, the size of the simulated cache will be limited to 8 cache lines divided in 4 sets, which is also one analogy used in the course COMP2120. The cache content will be displayed in a list numbered by set number on the left. Relevant content will be updated as the input programme proceeds, and the updated slot will be highlighted. Similarly, the size of the main memory simulated will be limited to 40 slots with its list display on the right, ordered by the simulated memory address.

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 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 8 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 and modules to be used in the simulation. Different objects initialized from their classes will have their own states and contents.

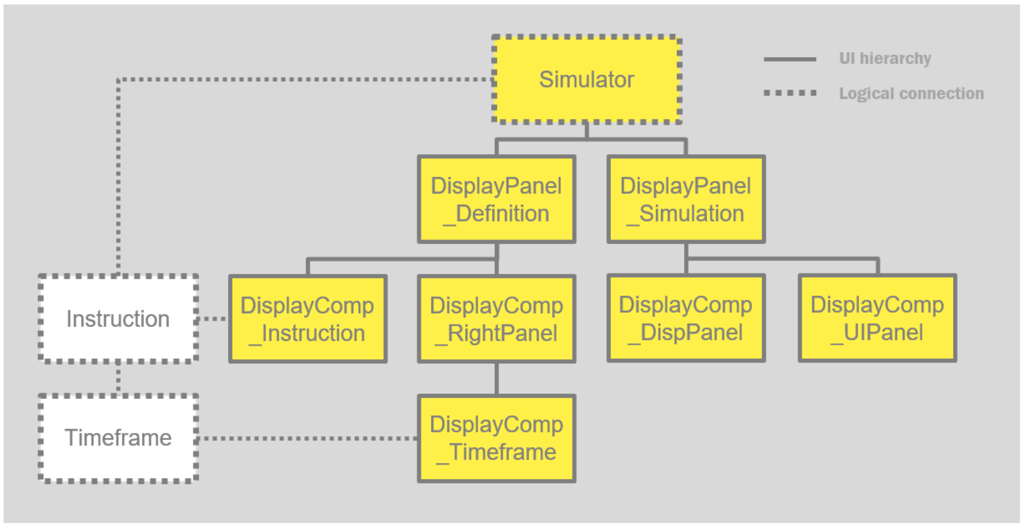


Figure 8. class structure of the CPU simulation

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.

In the display panel of instruction definition, instuction objects will be created per instruction item on the display, i.e. “DisplayComp_Instruction”. In other words, as graphical objects responsible for the user interface possess different behavior from the functional objects that actually model the corresponding components, the rule of thumb for design of the class relationship is that 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 5, containing the array of “DisplayComp_Timeframe” that correlate with the “Timeframe” objects for modelling the series of CPU operation cycles of the instruction.

The remaining classes in figure 8, 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 6. An image of the CPU architecture will serve 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.

By the time when this report is written, development of the simulation display part of the CPU simulation has only finished up to the UI. Therefore, as seen in figure 8 the classes for simulation display are responsible for UI properties and behaviour, denoted by the prefix “Display”.

Note that 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 9, 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 figure 8.

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 here is the Simulator class, which possesses dual properties as both a graphical object and a 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.

Cache Memory Simulation

The design of classes in the cache memory simulation follows the same paradigm used in the CPU simulation. Although development of this part has not yet started, ideas on implementation are reflected in the conceptual class structure shown in figure 9.

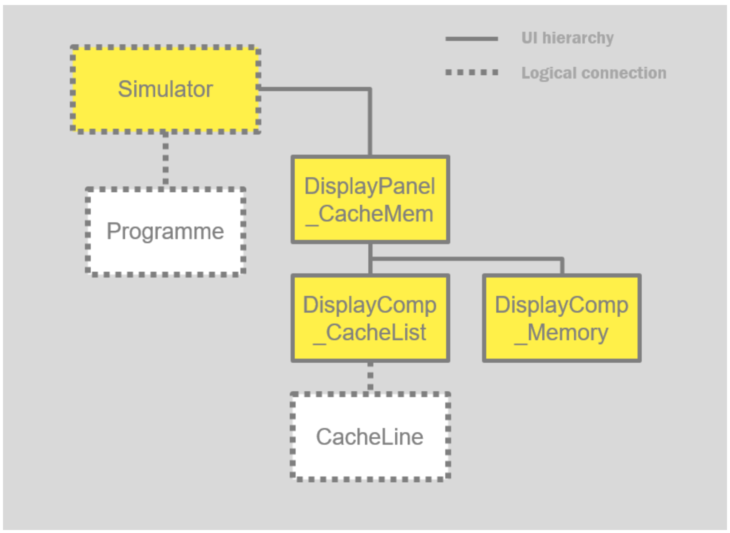


Figure 9. conceptual class structure of the cache memory simulation

There will be two major components in cache memory simulation. The first component concerns the cache, the second concerns the memory. The cache will be modelled as 8 “CacheLine” objects, each having its own index number, cache set affiliation and content etc. The 8 “CacheLine” objects together will be displayed by a single “DisplayComp_CacheList” object. This approach is a little different from the one-functional-object-to-one-graphical-controller practice as described above, because of the fact that each “CacheLine” object indeed has little information to be displayed, i.e. the index number, the data content and the updated state, and without any real-time interactive behaviour such as button. Therefore, it is sufficient to simply extract the variables in the “CacheLine” objects and collectively display them with one single graphical object.

The memory component will behave in the same fashion and is modelled similarly, using only one graphical controller to display the content of a list of memory objects. The Simulator class that will serve as the main window for the cache memory simulator contains the display panels of the cache and memory components, and will be initialized in the main function of the CPU simulation when the simulator runs.

The Simulator object in the cache memory simulation, when initialized, will contain the variable referencing the “Programme” object created in the previous CPU simulation, so as to capture the execution codes for the update of the cache content.

Note that the class structure diagram of the cache memory simulation is seemingly less complex than that of the CPU simulation. This is on one hand due to the expected simpler functionalities and implementation of the cache memory simulation, and on the other, the lack of practical evaluation of the design.

As the development of the cache memory simulation is placed in later stage of the project, which has not begun, the design illustrated above is mostly theoretical concept only. There may be updates to existing classes or addition of new classes where the practical development deems necessary.

Progress Review

By the time when this report is written, the development of the mnemonic instruction definition part of the CPU simulation has basically been completed, and the interface of the simulation display part has been finished with development on the functional components undergoing. Figure 10 and 11 show the respective demonstration.

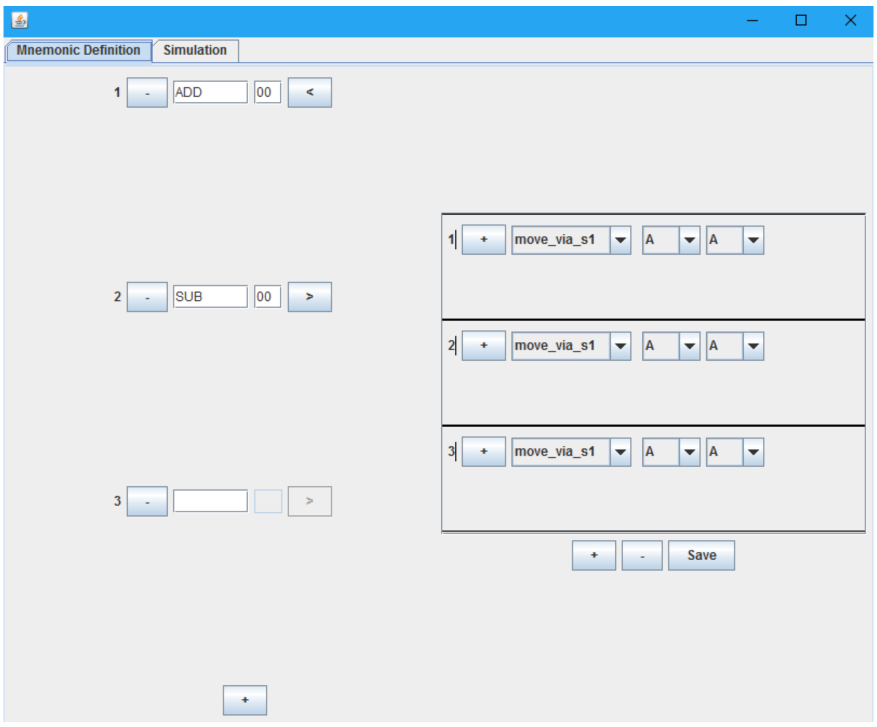


Figure 10. mnemonic instruction definition panel completed by far

The mnemonic instruction definition part has basically been completed. The instructions can be added on the left of the panel with symbol names and hexadecimal opcodes entirely defined by the users. Instructions are added by clicking the “+” button at the lower left while deletion of any instruction can be achieved by clicking the “-” button of the respective instruction. Index numbers of the instructions will update correspondingly.

The sequence of CPU actions can be added and configured on the right, with parallelization achievable by optionally adding the timeframes using the “+” button within the timeframes and initializing new timeframes by clicking the “+” button at lower right. The “Save” button will all the instructions to the respective logical objects and possibly be utilized in export the definition in future development. But the “Save” button is currently not functional while all instructions are saved at the moment when any changes are made. The “-” button on lower right will remove the timeframe(s) with the last index number and individual removal of any timeframe is currently not possible for the sake of reducing implementation complexity.

For the simulation display panel as shown in figure 11, all the implementations are currently on the UI level but not functional. Functionalities such as accepting user programme input, decoding the programme, executing the programme and visualizing the simulation are proceeding. A glimpse of the expected work is provided in the display section of figure 11, where a red dot signaled “test” is displayed, mimicking the data flow simulation when the backend logic of this part is completed.

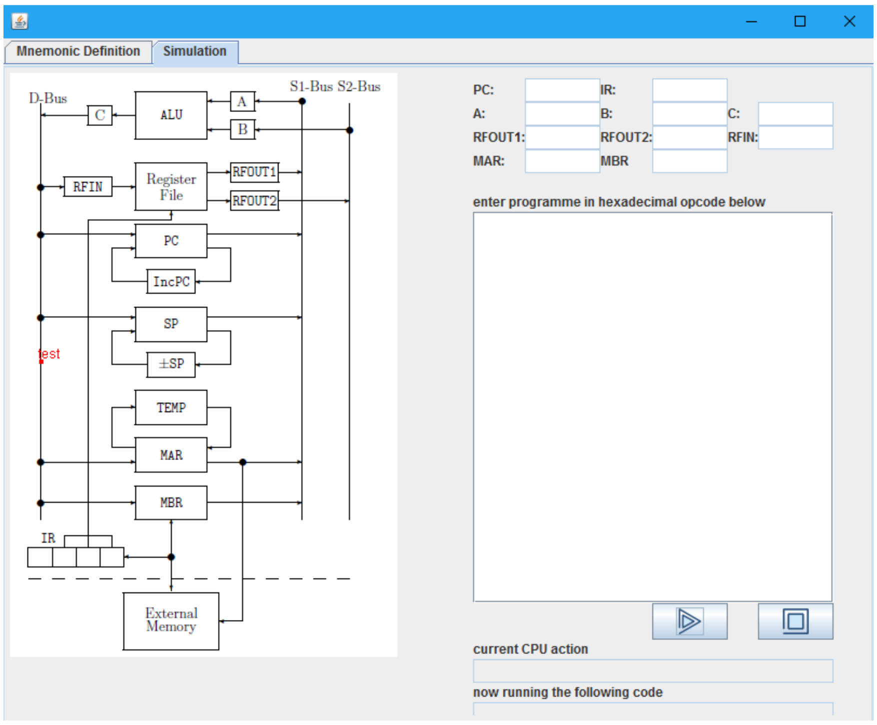


Figure 11. simulation display panel completed by far

Besides the expected and unfinished components, there are certainly improvements that can be made on currently available progress. The most significant problem with this deliverable is with the user interface. With careful observation from figure 10, one will notice that layout and spacing of the instruction components on the left and the right list of CPU actions are not entirely comfortable. They should be more compact and organized. But these improvements on the layout shall be catered after finishing the functional parts of the expected components to ensure the simulator is at least sufficiently functional by the end of this project.

The project has by far proceeded according to the schedule defined earlier in the project plan. With the above suggestions, the development of the project will continue with the completion of the CPU simulation expected by February 2017 and the development of the cache memory simulation following.

Development of cache memory simulation is placed in a rather late schedule because of a few reasons. Firstly, it is anticipated to be simpler as compared with that of CPU simulation as the simulation in this part is basically display of data without a large extent of in-depth user customization and interaction as the instruction definition parts does; secondly, the basic infrastructure will be completed in the part of CPU simulation, for example, the programme decode and simulation will be established already in CPU simulation and the rest in memory simulation is only to retrieve useful data out of the process, which is less development needed; thirdly, as communicated with the project supervisor, cache memory simulation possesses a lesser importance and priority in the project as compared to CPU simulation, thus the arrangement of developing in later stage after that of CPU simulation has been finished.

Table 2 summarizes the progress thus far and outlines future development plan.

Completed	
1. Instruction Definition of CPU Simulation	
2. UI of CPU Simulation	
To be expected	
Date	**Event**
Feb 2017	Functionality of Simulation Display of CPU Simulation
Feb – Apr 2017	Cache Memory Simulation

Table 2. progress summary and development timeline

Limitations

The simulation of a computer system is not an easy task. Despite the carefully chosen scope and reduced complexity of this project, there are challenges.

The major technical challenge is with CPU actions parallelization in the CPU simulation. The customized mnemonic definition is itself challenging in a sense that there are a vast number of action sequences resulting from the different combinations of CPU actions freely added and selected by the student users. It is near impossible to simulate all possible sequences in advance, capture and thus avoid errors. This problem, fortunately, can be minimized systematically by analyzing the compatibility of consecutive CPU actions selected in sequential order.

Nonetheless, the parallelization process adds to the degree of freedom of the possible CPU action sequences and increases the difficulty of error handling.

A solution to this is to pose some limitations on the process. For example, the CPU actions now are predefined where users can only choose from but not create their own. There are also limits on the number of instructions and CPU operation timeframes that can be added. All this prevents the complexity of the sequence definition to increase indefinitely and limits the growth of error rate to an acceptable and avoidable scale.

Another major challenge is with user acceptance. This is an educational project and the primary users are students taking the course COMP2120 Computer Organization. It will be ideal to include the students as test users for user acceptance test in the process of development to evaluate the performance of the simulator.

However, due to conflict of the development timeline of this project and the timetable of the course, it is unlikely that any student joining the course can act as a test user.

A workaround to the problem is to include the supervisor and the teaching assistants of the course in the process of development. Workable iterations will be sent to them upon completion for iterative feedback and improvement. Regular meetings with the supervisor has been set up in a bi-weekly manner and comments of improvement have been received and refinements made accordingly.

References

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