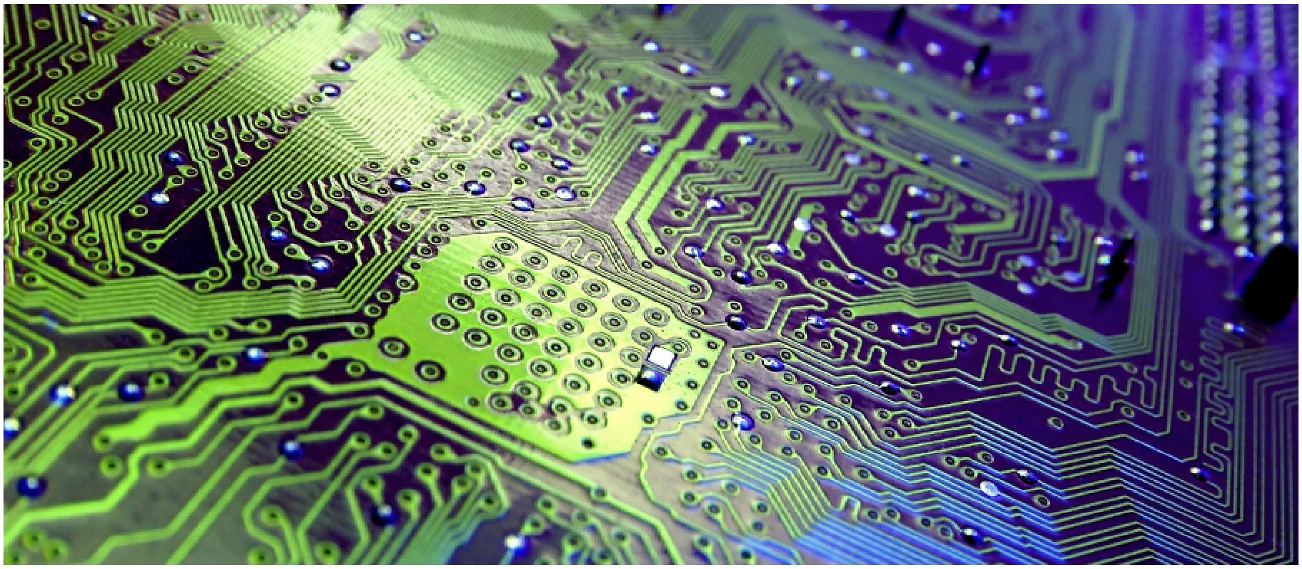
CSIS0801 FINAL YEAR PROJECT

PHASE 3 DELIVERABLE

FINAL REPORT

TOPIC: COMPUTER SYSTEM SIMULATOR



WONG JING HIN (KENT)

UID: 3035051060

Table of Contents

Introduction

This project aims to develop a simulator for a computer system based on a simple instruction set that simulates the instruction execution process, cache memory and memory hierarchy for teaching purposes with the following features:

Interconnection of different Operational Units

The data movements and transformations occurring in the processors, cache memory and main memory will be covered in instruction set simulation. Users will be able to see the changes of all these operational units in the execution and the interactions among them.

Graphical User Interface (GUI)

The simulator of this project was developed with GUI. Components of the processor (e.g. registers & program counter), cache memory and main memory are displayed on the screen in form of graphic. There are 3 benefits of GUI for the simulator, firstly the structure of the computer system and interconnections among components can been shown clearly; also the data movements and transformation in different components and instruction execution can be more effectively illustrated; students will also be able to view the full picture of instruction set execution process easier than before as the whole relevant architecture is displayed in the output screen, they can see which components are affected and which are not affected.

Flexible Configuration

Core configurations are independent of the core program source code. When a user runs the simulator, the simulator automatically reads the external configuration file to set necessary configuration for simulation, after that it will receive user's input of the storage path of the instruction set binary file for the simulation. Users only need to amend the configuration file and then re-execute the simulator program if they want to do the simulation in different configurations.

Project Background

Current Situation

Computer Organisation is a core course for undergraduates of Computer Science Major, which covers the study of operational units of a computer that are involved in Instruction Execution Cycle (e.g. CPU) and their interconnections. There are several kinds of those operational units which further contains a number of components in each of them, for example in a CPU, there are different types of registers as well as ALU inside.

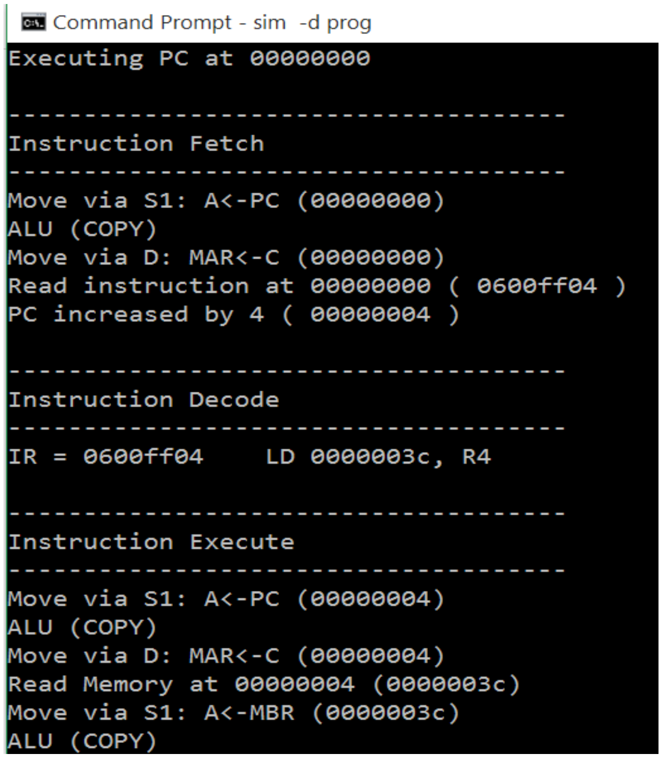
In an instruction execution, there are lots data movements among different components and transformations within components. Moreover, components are connected with each other differently, forming a complicated architecture of a computer system. Therefore, lots of undergraduate students find the flow and relevant concepts difficult to understand just based on verbal descriptions by lecturer and on lecture note or textbooks.

Existing Teaching Aids

In order to explain the concepts and the data flow among components of computer system, there are 2 mini simulators available as teaching aids (possibly jointly developed by the lecturer and tutor) for the Computer Organisation course.

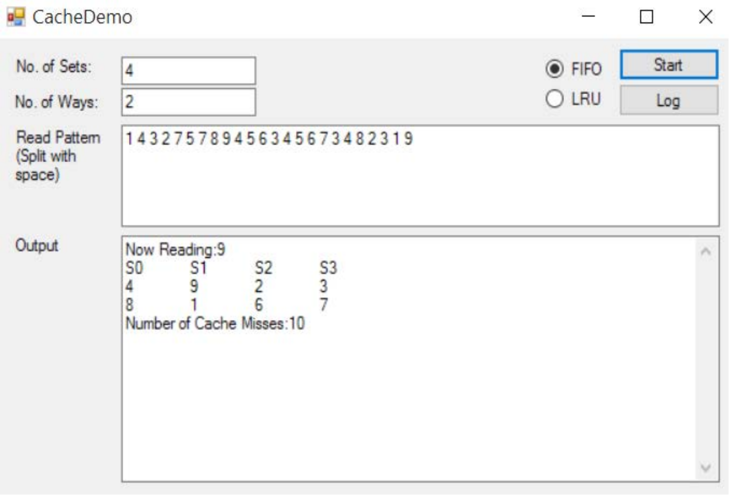
The first one is a program written in C++ programming language which simulates the data flow in Control Unit (CU) and Arithmetic and Logic Unit (ALU) of CPU. This program has to be run in command-line interpreter with a binary instruction set text file as input, the sequence of data movement and transformation of every instruction will be displayed as pseudocode-like descriptions on the console output.

The screen capture on the right is the sample output of this CPU simulating program:



```
Executing PC at 00000000

-------------------------------------
Instruction Fetch
-------------------------------------
Move via S1: A<-PC (00000000)
ALU (COPY)
Move via D: MAR<-C (00000000)
Read instruction at 00000000 ( 0600ff04 )
PC increased by 4 ( 00000004 )

-------------------------------------
Instruction Decode
-------------------------------------
IR = 0600ff04    LD 0000003c, R4

-------------------------------------
Instruction Execute
-------------------------------------
Move via S1: A<-PC (00000004)
ALU (COPY)
Move via D: MAR<-C (00000004)
Read Memory at 00000004 (0000003c)
Move via S1: A<-MBR (0000003c)
ALU (COPY)
```

Another one is a cache memory simulator written in C#. It is a small program simulating mapping and replacement algorithm of cache memory with Graphical User Interface (GUI). The screen capture on the right shows a sample run of this cache memory simulator:



No. of Sets: 4
No. of Ways: 2
● FIFO ○ LRU
Read Pattern (Split with space): 1 4 3 2 7 5 7 8 9 4 5 6 3 4 5 6 7 3 4 8 2 3 1 9

Output

```
Now Reading:9
S0      S1      S2      S3
4       9       2       3
8       1       6       7
Number of Cache Misses:10
```

Problems and Proposed Solution

There are several major drawbacks of the above existing teaching aids. Firstly, the two operational units of computer system architecture are simulated separately, it failed to show full and real picture of the operation of an instruction set in the computer system. Also, for the CPU simulating program, all configurations such as memory size, register file size and operation representing code (e.g. 00000101 as MOV operation) are "hardcoded" in the program source code, direct amendments on the program code is needed if we want to perform the simulation in different configurations. In addition, for the first simulating program, although the data movements and transformations are shown in sequence, it is shown in form of "sentences" in command line prompt, not only the full picture of the CPU operation cannot be shown (as only content of affected components are displayed), command line output format is also unattractive to read. Moreover, the output has low readability due to command-line form display, there are chunks of statements displayed on the screen for every instruction execution, which contributes a reducing clearness on the illustration of the flow and relevant concepts, thus students still find it difficult to learn the computer system operations even with these teaching aids.

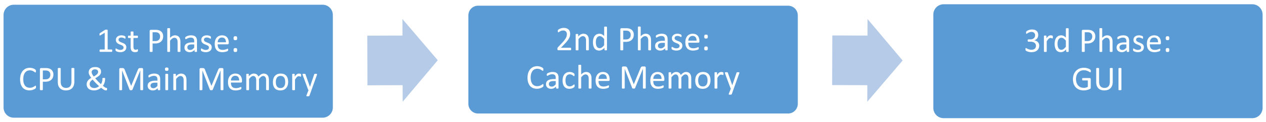
As a "user" using these teaching aids before, I appreciate the efforts of developing such programs but unfortunately it has to be admitted that these programs are not effective enough in aiding students to learn computer organization (still they are good references). Thus, developing a new integrated computer system simulator which can cover more operational units, more functionalities and with greater flexibility in configurations is undeniably desirable and meaningful.

Project Methodology

Phased methodology was adopted for this project.

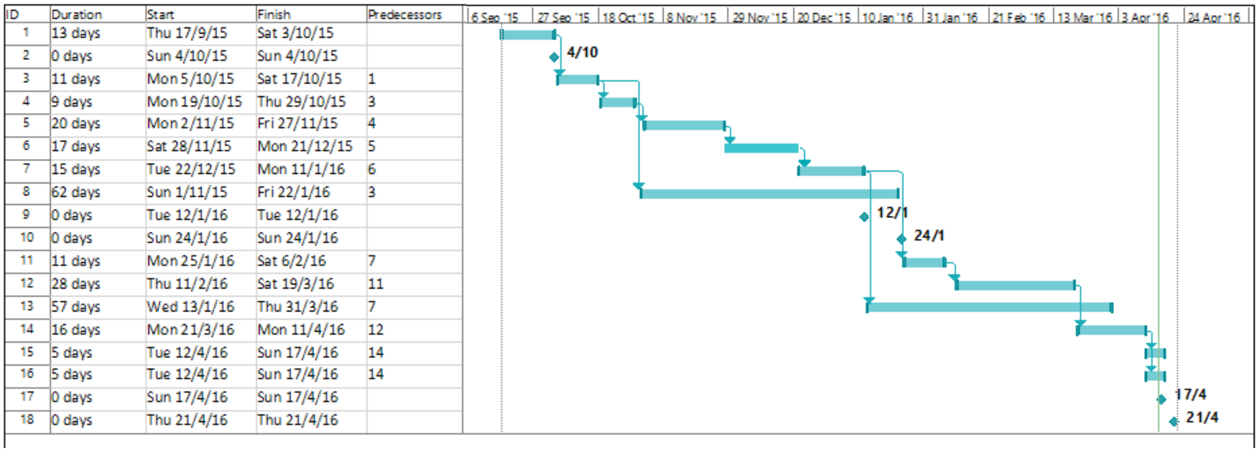
The system architectural design of the simulator will be complicated as it covers not only one operational units in computer system, which contains sub-components and interrelates among themselves differently. At the same time, CPU is the core part of instruction set execution process, the functionalities of the CPU part should be completed before integrating further operational units to the simulator. Therefore, the development of the simulator should be carried out in a unit-by-unit approach, and the CPU part and Cache Memory part is an end-to-finish relationship. Therefore phased development methodology will be best-fit for this situation. The main advantage of phased development is that it allows developers to put focus on each feature one by one that scopes are well defined under each phase, hence the development burden can be eased and quality of each feature of the system can be ensured.

The development cycle of the simulator was categorized into 3 phases. CPU part was developed in 1st phase and cache memory part was developed in 2nd phase, in which the order was in the other way round of the accessing order of operation units in an instruction set execution (i.e. accessing order is: cache memory -> CPU, so development order was: CPU -> cache memory). The deliverable from 1st Phase was an executable program that can simulate instruction set execution that without the use of cache memory. 2nd phase development was an extension of the deliverable from 1st phase, cache memory and related operations were added on top of the existing hierarchy of the simulator program of 1st phase deliverable. The development of the first 2 phases was in command-line output, when the 2nd phase was completed, the core functionalities of the simulator was considered as implemented with correct logic flow. Therefore output format was transformed into GUI form as the final phase of the project.



Project Timeline

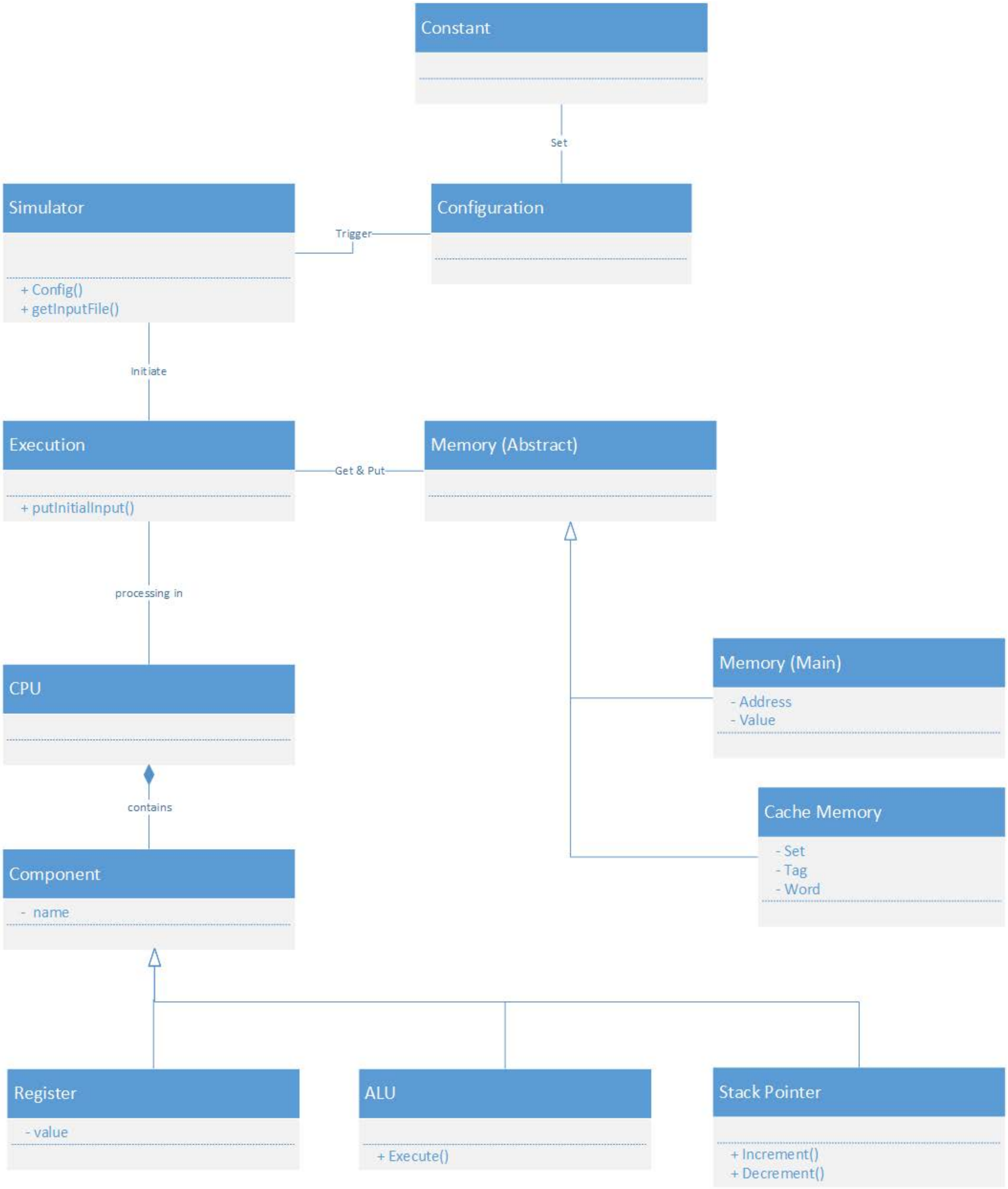
ID	Task Name	Start Date	End Date	Predecessors
1	Requirements Gathering & Analysis	17/9/15	3/10/15	
2	**Milestone: Project Plan Submission**	**4/10/15**	**n/a**	
3	Preliminary Study & Research	5/10/15	17/10/15	1
4	Phase 1 (CPU) Analysis & Design	19/10/15	29/10/15	3
5	Phase 1 Coding & Testing I	2/11/15	27/11/15	4
6	Break for Preparing Final Examinations	28/11/15	21/12/15	5
7	Phase 1 Coding & Testing II	22/12/15	10/1/16	6
8	Documentation for Phase 1	1/11/16	24/1/16	3
9	**Milestone: First Presentation**	**12/1/16**	**n/a**	
10	**Milestone: Interim Report Submission**	**24/1/16**	**n/a**	
11	Phase 2 (Cache Memory) Analysis & Design	25/1/16	6/2/16	7
12	Phase 2 Coding & Testing	11/2/16	19/3/16	9
13	Phase 2 Documentation	30/1/16	17/4/16	7
14	Phase 3 (GUI) Implementation	21/3/16	11/4/16	12
15	Integrated Testing & Debugging	12/4/16	17/4/16	14
16	Phase 3 Documentation	12/4/16	17/4/16	12
17	**Milestone: Final Report Submission**	**17/4/16**	**n/a**	
18	**Milestone: Final Presentation**	**20/4/16**	**n/a**	



ID	Duration	Start	Finish	Predecessors
1	13 days	Thu 17/9/15	Sat 3/10/15	
2	0 days	Sun 4/10/15	Sun 4/10/15	
3	11 days	Mon 5/10/15	Sat 17/10/15	1
4	9 days	Mon 19/10/15	Thu 29/10/15	3
5	20 days	Mon 2/11/15	Fri 27/11/15	4
6	17 days	Sat 28/11/15	Mon 21/12/15	5
7	15 days	Tue 22/12/15	Mon 11/1/16	6
8	62 days	Sun 1/11/15	Fri 22/1/16	3
9	0 days	Tue 12/1/16	Tue 12/1/16	
10	0 days	Sun 24/1/16	Sun 24/1/16	
11	11 days	Mon 25/1/16	Sat 6/2/16	7
12	28 days	Thu 11/2/16	Sat 19/3/16	11
13	57 days	Wed 13/1/16	Thu 31/3/16	7
14	16 days	Mon 21/3/16	Mon 11/4/16	12
15	5 days	Tue 12/4/16	Sun 17/4/16	14
16	5 days	Tue 12/4/16	Sun 17/4/16	14
17	0 days	Sun 17/4/16	Sun 17/4/16	
18	0 days	Thu 21/4/16	Thu 21/4/16	

Design and Implementation

Program Class Design



The program class structure simulates the real structure of different parts in instructions execution.

The Simulator itself is an object. Running the Simulator will trigger configuration, constants will be set from the external configuration property file read by the program.

The Simulator can initiate an execution of instruction set, thus there is another class for an execution. The execution involves processing and storage, thus there are 2 class, CPU & Memory, expanded from the Execution class.

For Memory class, there are 2 types of memory involved, thus Main Memory & Cache Memory class object are created, which extends the Memory (Abstract) Object. However as the Main Memory and Cache Memory behaves significantly differently and does not share any common procedure, thus the Memory (Abstract) object was not created in the real implementation.

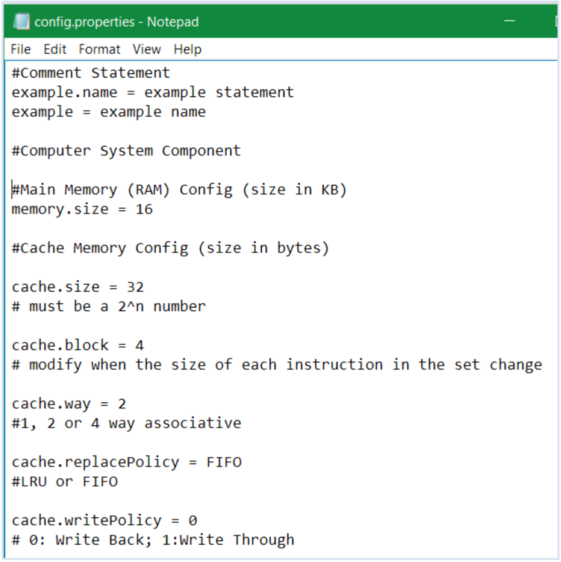
CPU comprises of different type of components, including registers (e.g. register files & program counter) in Control Unit (CU) and the Arithmetic & Logic Unit (ALU). Stack Pointer class is created independent of Register object (but still extends from Component object as their nature are the same) due to its different structure and usage compared with other registers in CU.

Configuration Files

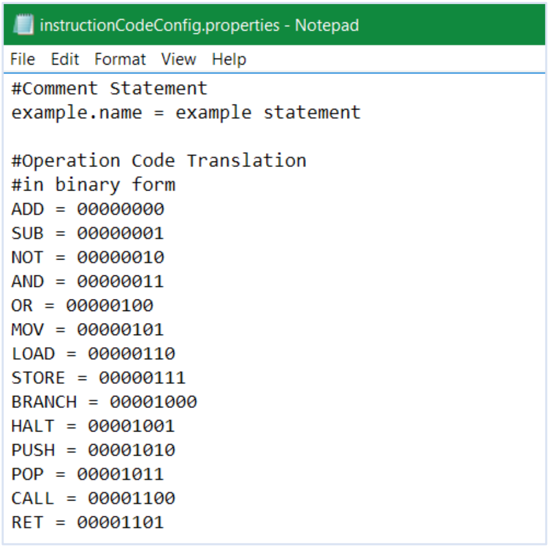
There are 2 configuration files, namely “instructionCodeConfig.properties” and “config.properties”, which are external text files (can be opened using common text editor) that can be modified by any user.

“config.properties” contains some general configuration for running the simulator, mainly for cache memory operation, e.g. cache size, cache memory size, number of way associative, replacement algorithm.

“instructionCodeConfig.properties” contains definition of operation code, e.g. ADD, BRANCH, HALT. Users can change the binary code representing the operation.



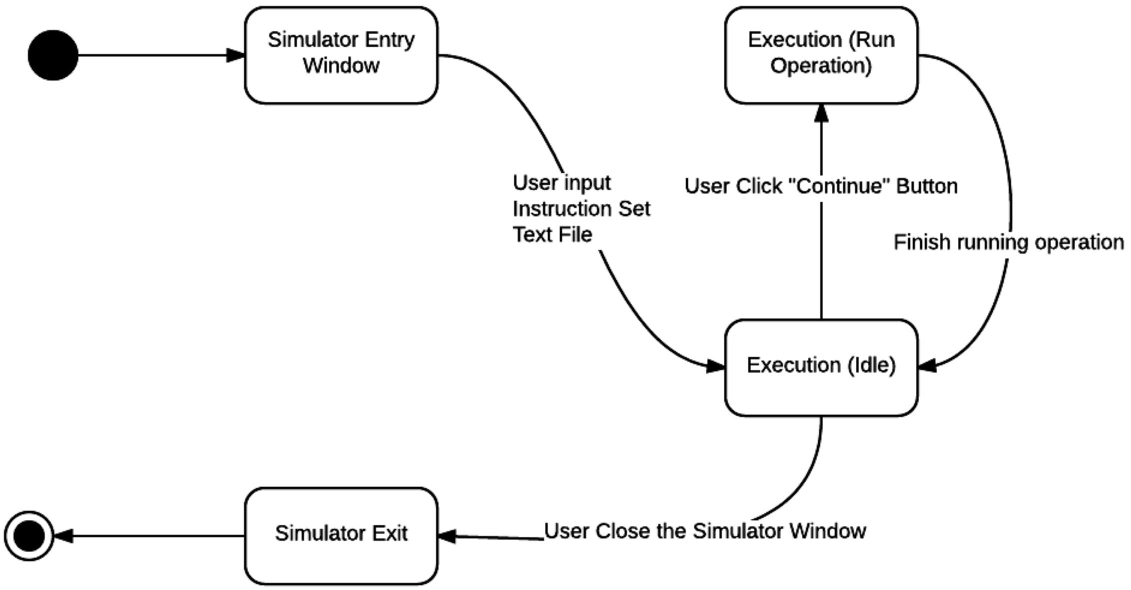
```
#Comment Statement
example.name = example statement
example = example name

#Computer System Component

#Main Memory (RAM) Config (size in KB)
memory.size = 16

#Cache Memory Config (size in bytes)

cache.size = 32
# must be a 2^n number

cache.block = 4
# modify when the size of each instruction in the set change

cache.way = 2
#1, 2 or 4 way associative

cache.replacePolicy = FIFO
#LRU or FIFO

cache.writePolicy = 0
# 0: Write Back; 1:Write Through
```


instructionCodeConfig.properties - Notepad
File Edit Format View Help

```
#Comment Statement
example.name = example statement

#Operation Code Translation
#in binary form
ADD = 00000000
SUB = 00000001
NOT = 00000010
AND = 00000011
OR = 00000100
MOV = 00000101
LOAD = 00000110
STORE = 00000111
BRANCH = 00001000
HALT = 00001001
PUSH = 00001010
POP = 00001011
CALL = 00001100
RET = 00001101
```

Program Flow

The simulator program flow can be illustrated by the following State-Machine Diagram:


Simulator Entry Window
Execution (Run Operation)
User input Instruction Set Text File
User Click "Continue" Button
Finish running operation
Execution (Idle)
Simulator Exit
User Close the Simulator Window

After start running the Simulator (Simulator Entry Window State)

The main method in the Simulator will run, new configuration class will be created which trigger the configuration setting for the simulator and execution.

Simulator Entry Window will be created and ask user to input the file path of the instruction set to be demonstrated.

Execution (Idle) State

After reading the file path, the simulator will read the external text file and set the content as a List of Memory object. Then the creation of Execution class will be automatically triggered. In constructor of the Execution class, there will be creation of processor object and memory object, the content of these objects will be set according to the general configuration and the List of Memory object parsed when constructing the new Execution object.

The entry window of the simulator will be disposed and replaced by the Execution window. The Execution window will stay idle until user perform action (click “Continue” button) to trigger the start of the instruction set execution.

Execution (Run Operation) State

When the simulator is triggered to start, the Simulator will start execute the next instruction. The Simulator will go back to Execution (Idle) State when the execution of operation of the instruction is completed, to wait for user’s action to trigger running next instruction. If the instruction executed is a HALT instruction, user will no longer be able to click the “Continue” button as there are no next instruction to run, the Simulator will stay in Execution (Idle) State.

Simulator Exit State

Instead of start/continue next instruction execution, user can choose to exit the Simulator program, when the user close the Simulator Execution window in the Execution (Idle) State, the Execution window will be disposed and the Simulator program will terminate.

Instruction Execution Flow

For each instruction execution in the Simulator, the algorithm quite resembles a real instruction execution. Firstly the "Program Counter" object in the Processor object will be analyzed, to obtain the respective memory address of the instruction or data to be read, and the "PC" value will be incremented by 4, which is very alike to the "Instruction Fetch" of real instruction execution. The textbox on the right shows the pseudocode of this procedure.

```
Execute_Instruction() {
        Address <- PC.value
        Instruction <- Memory[Address]
        PC.value <- PC.value + 4
        Analyze_Instruction(Instruction)
}
```

Next will be instruction decoding. The content stored in "Instruction Register" object will be extracted and analyzed. The string will be divided into 4 sub-string, the 1st sub-string indicate the operation code, the program will then map the operation code with the "definition table" set from configuration, and then continue to analyze the other 3 sub-string (the way to analyze is defined differently for different operation).

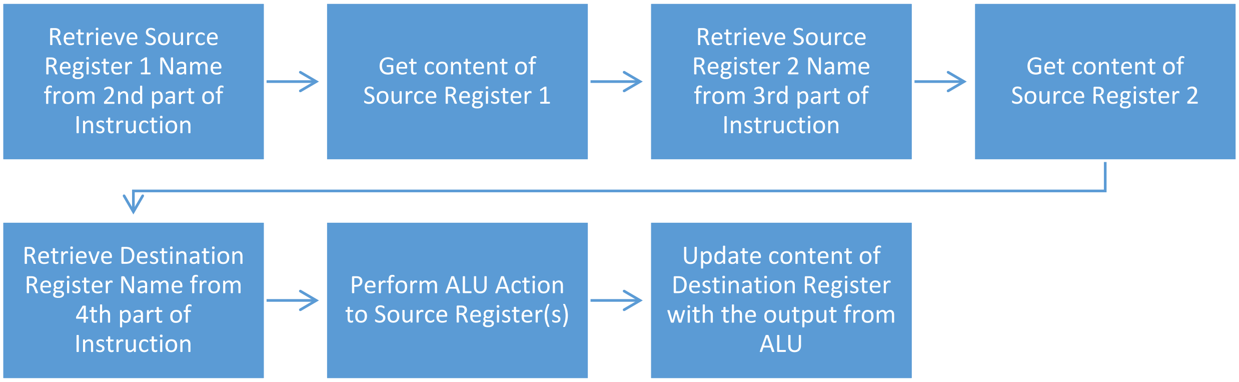
The mapping of operation code was implemented as a case statement, different procedure will be executed for different operation code mapped. The execution will continue with the mapped procedure (methods).

```
Analyze_Insturction(Insutuction) {
SepOp <- DivideOperationCode(Instruction)
operationCodeStr <- SepOp.substring(1st Part)
case (operationCodeStr) of
begin
        "00000000" : add();
        "00000001" : sub();
        "00000010" : move();
        ……
        "00001001" : halt();
end
}
```

Operation Algorithm Design

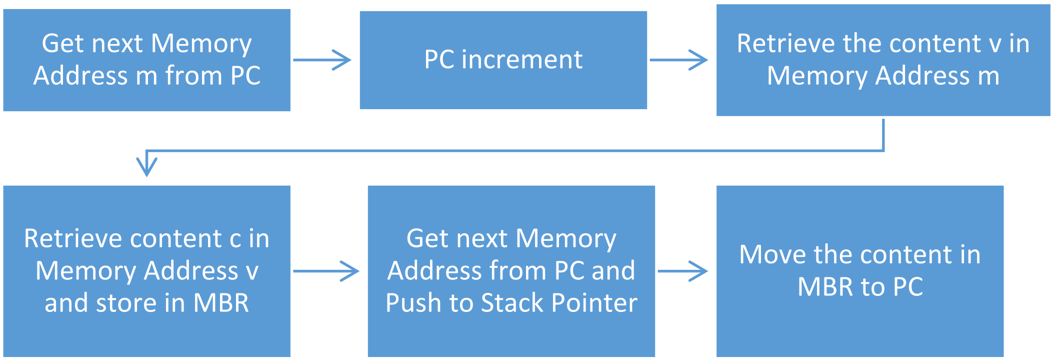
The algorithm of each operation will be illustrated by respective flow diagrams:

Add/Sub/And/Or/Not/Move Operation

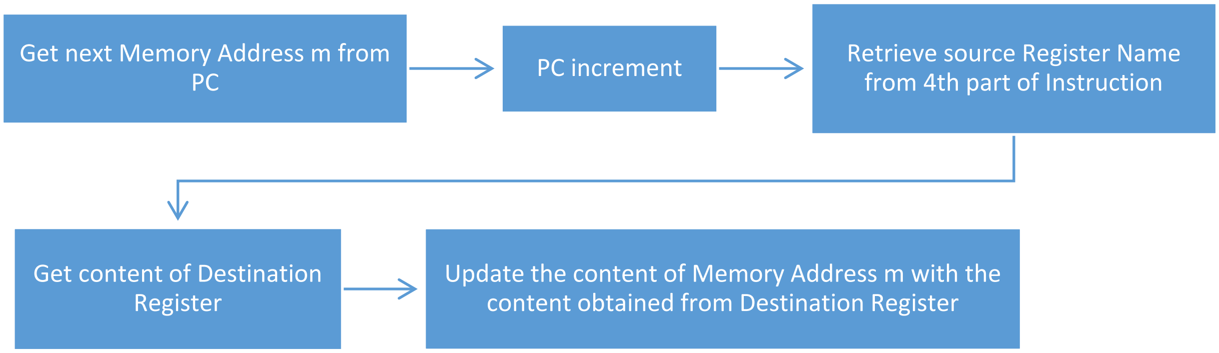

Retrieve Source Register 1 Name from 2nd part of Instruction
Get content of Source Register 1
Retrieve Source Register 2 Name from 3rd part of Instruction
Get content of Source Register 2
Retrieve Destination Register Name from 4th part of Instruction
Perform ALU Action to Source Register(s)
Update content of Destination Register with the output from ALU

The algorithm of these operations are similar except that for “Not” and “Move” there will be no lines involving Source Register 2.

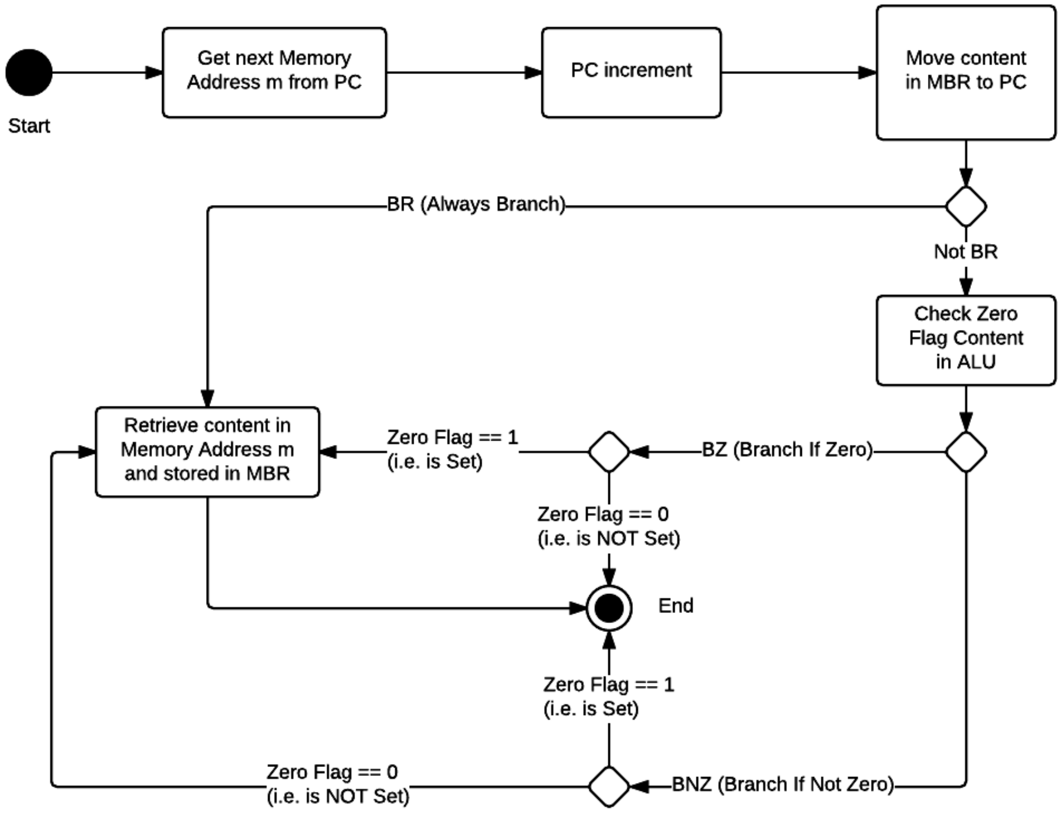
Load Operation


Get next Memory Address m from PC
PC increment
Retrieve the content v in Memory Address m
Retrieve content c in Memory Address v and store in MBR
Get next Memory Address from PC and Push to Stack Pointer
Move the content in MBR to PC

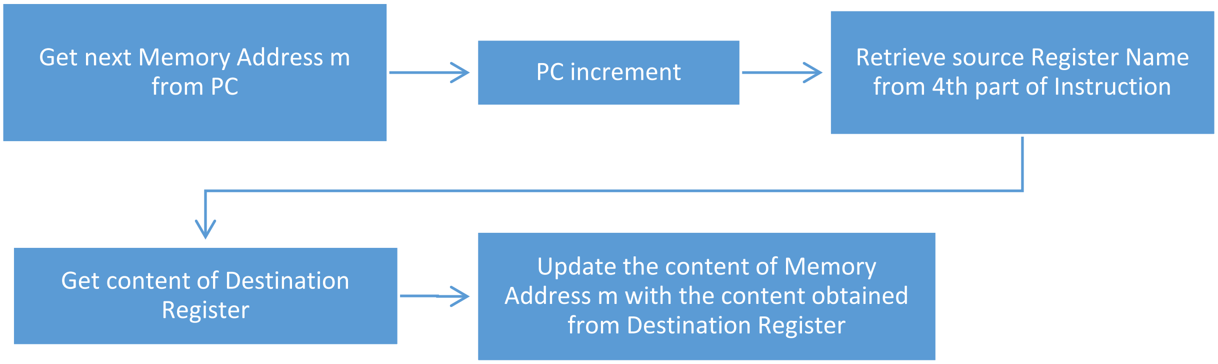
Store Operation


Get next Memory Address m from PC
PC increment
Retrieve source Register Name from 4th part of Instruction
Get content of Destination Register
Update the content of Memory Address m with the content obtained from Destination Register

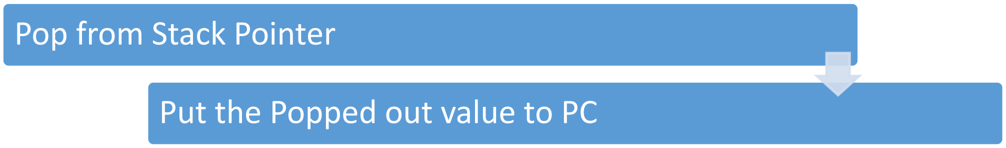
Branch Operation


Start
Get next Memory Address m from PC
PC increment
Move content in MBR to PC
BR (Always Branch)
Not BR
Check Zero Flag Content in ALU
Retrieve content in Memory Address m and stored in MBR
Zero Flag == 1 (i.e. is Set)
BZ (Branch If Zero)
Zero Flag == 0 (i.e. is NOT Set)
End
Zero Flag == 1 (i.e. is Set)
Zero Flag == 0 (i.e. is NOT Set)
BNZ (Branch If Not Zero)

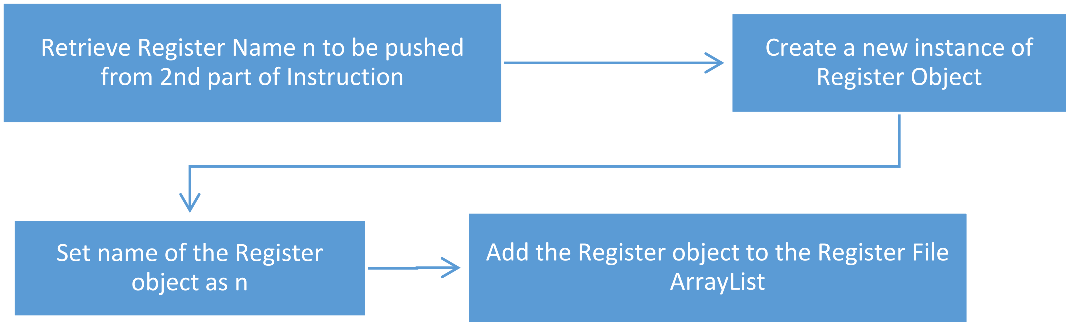
Call Operation (of Function Call)


Get next Memory Address m from PC
PC increment
Retrieve source Register Name from 4th part of Instruction
Get content of Destination Register
Update the content of Memory Address m with the content obtained from Destination Register

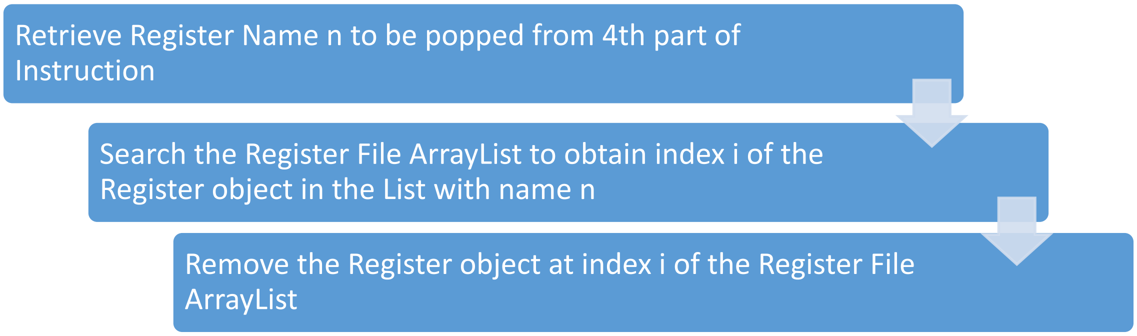
Return Operation (of Function Call)



Push Operation (of Register File)


Retrieve Register Name n to be pushed from 2nd part of Instruction
Create a new instance of Register Object
Set name of the Register object as n
Add the Register object to the Register File ArrayList

Pop Operation (of Register File)


Retrieve Register Name n to be popped from 4th part of Instruction
Search the Register File ArrayList to obtain index i of the Register object in the List with name n
Remove the Register object at index i of the Register File ArrayList

Cache Memory

Overview

The Cache Memory part of the Simulator supports 2 replacement policies, namely “First-In-First-Out” (FIFO) and “Least Recently Used” (LRU), as well as 2 write policies, namely “Write Back” and “Write Through”, which can be modified in the external configuration file.

Blocks in Cache Memory are divided into different sets. The Simulator program support x-way associative organization of Cache, i.e. containing x cache line for each set. x should be a number equal to 2^n.

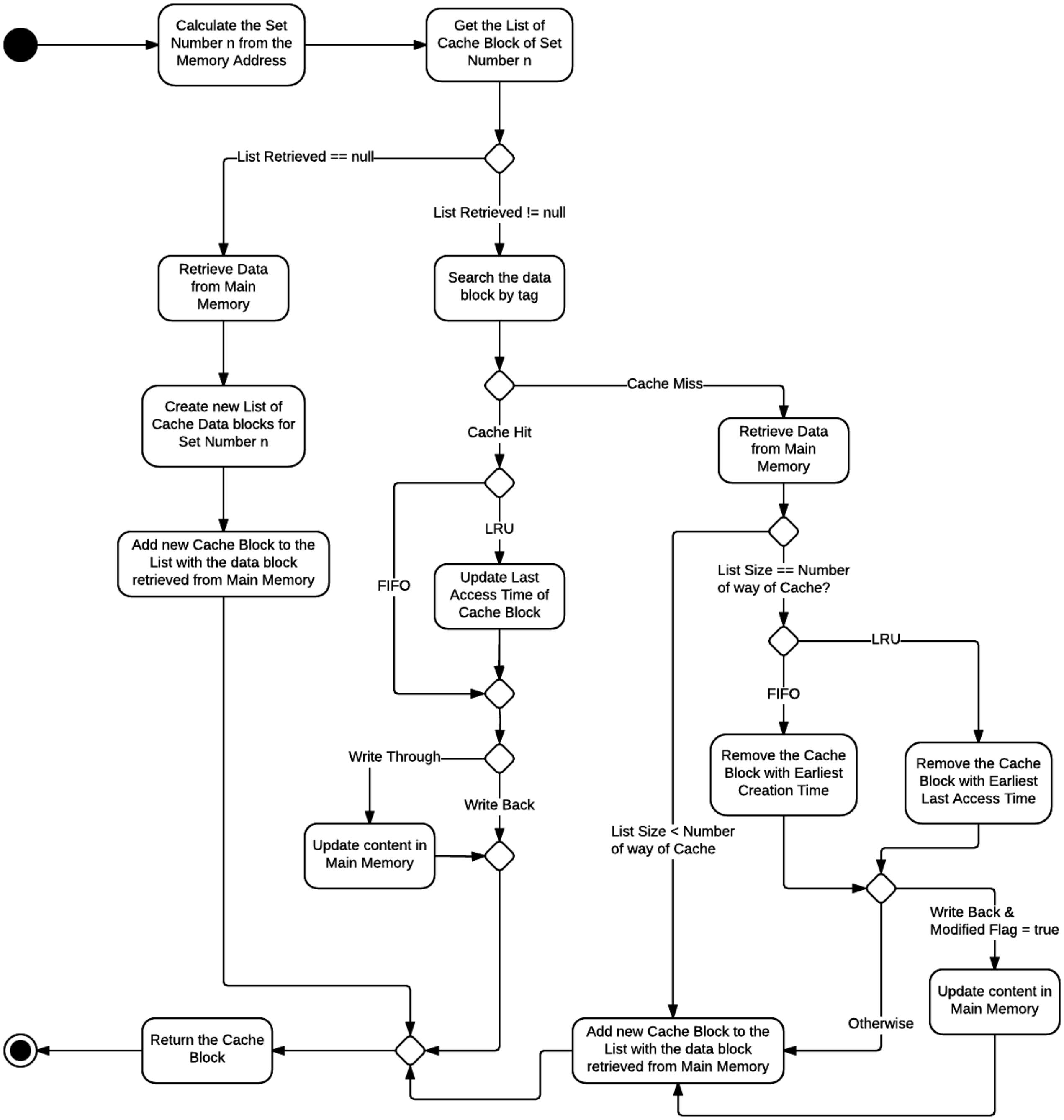
Cache Memory consists of Cache Memory blocks which are categorized with a Set Number, which can be easily obtained by retrieving rightmost n bits of the Memory Address in binary form for x-way associative Cache given that x is equals to 2^n.

There are 4 variables in a Cache object, which is shown below with their respective use:

Variable	Use
Tag	Unique identification tag for the Cache entry in a Set
Word	Data stored in the Cache Block
Timestamp	Creation Time (for FIFO) or Last Access Time (for LRU)
Modified Flag	Set true if the data stored in the Cache Block has been modified

For FIFO, when replacement of Cache Block has to be occurred, the Cache Block with earliest creation time should be replaced. While for LRU, the Cache Block with earliest last access time should be replaced. Timestamp will be used to find out the earliest creation time or earliest last access time.

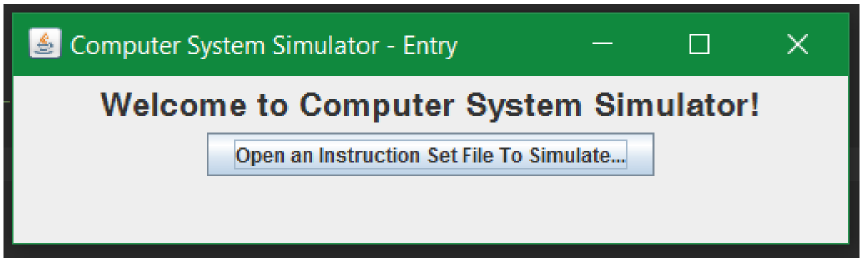
Flow Diagram of Cache Procedure


Calculate the Set Number n from the Memory Address
Get the List of Cache Block of Set Number n
List Retrieved == null
List Retrieved != null
Retrieve Data from Main Memory
Search the data block by tag
Create new List of Cache Data blocks for Set Number n
Cache Miss
Cache Hit
Retrieve Data from Main Memory
LRU
Add new Cache Block to the List with the data block retrieved from Main Memory
FIFO
Update Last Access Time of Cache Block
List Size == Number of way of Cache?
LRU
FIFO
Write Through
Remove the Cache Block with Earliest Creation Time
Remove the Cache Block with Earliest Last Access Time
Write Back
Update content in Main Memory
List Size < Number of way of Cache
Write Back & Modified Flag = true
Update content in Main Memory
Otherwise
Return the Cache Block
Add new Cache Block to the List with the data block retrieved from Main Memory

GUI

Simulator Entry Window

The entry window of the Simulator is a small dialogue box, displaying a welcome message and also a button, for user to specify the file path of the instruction set binary text file. The button was implemented using JFileChooser of Java AWT library, which provide an interface to let user find the file using the directory system, so that user no need input the file path to the Simulator by themselves.



Computer System Simulator - Entry

Welcome to Computer System Simulator!

Main Window (Demonstrating Execution)

The main window is divided into 5 part, with the use of border-pane. The structure of the border-pane can be illustrated as follows:

<table>
<tr><td colspan="3">top: Description Pane</td></tr>
<tr><td rowspan="2">left: Canvas</td><td>center:
Cache Memory Table</td><td rowspan="2">right:
Main Memory Table</td></tr>
<tr><td>center:
Registers Table</td></tr>
</table>

Description Pane contains the "message" and the "Continue Button". "Message" is the description of the current process in the instruction execution, like data move from MBR to MAR, or informing users for existence of cache miss & cache hit. "Continue Button" is located at the rightmost of the Description Pane, user just need to click the button once to trigger the execution of next instruction. The button will have been disabled when the execution of an instruction is taken place, and will be "re-opened" to users when the operation ends.

Canvas is the part that containing all necessary components in an instruction set execution in a simulation. The components, e.g. PC, MAR, MBR are added to the canvas as Label, and will be highlighted when they are involved. The bus connecting each component are added to the canvas as PolyLine object, they will also be highlighted when they are involved.

The center pane is further divided into 2 sub-division, displaying the data block in Cache Memory and content of some important registers and register files.

The right pane displays the content of main memory, i.e. the imported instruction set.

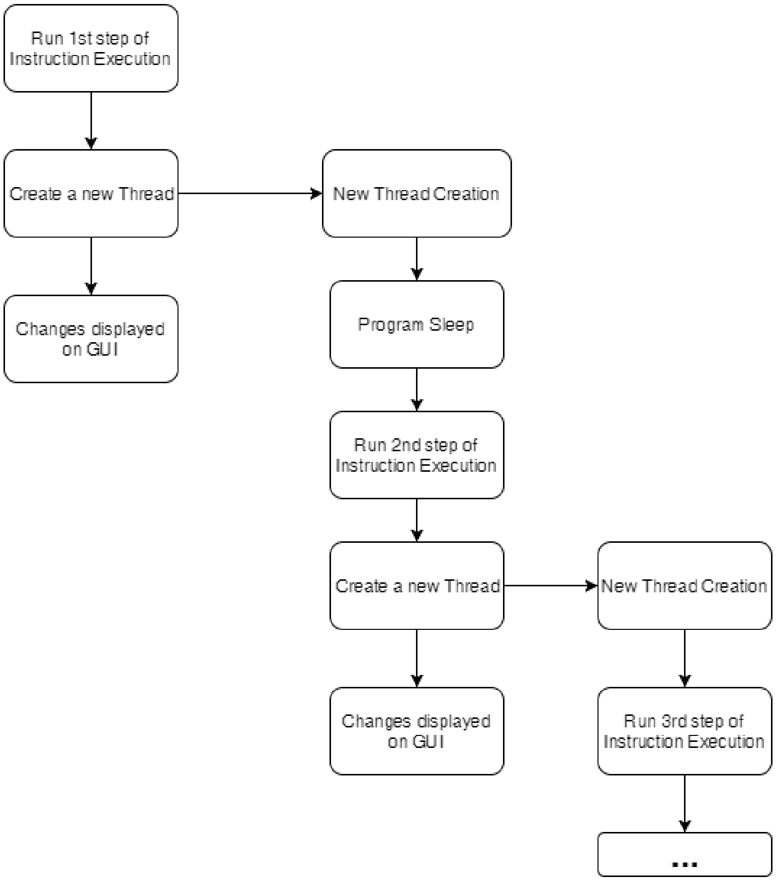
Under this layout design, content of memory and registers are shown all the time throughout the whole execution simulation.

Use of JavaFX Library

JavaFX instead of Java AWT or Swing Library is chosen for the implementation of the GUI feature of the simulator for several reasons. Firstly, the JavaFX API is more consistent across components, which is significant as in the simulator different components will have to apply similar type of changes, e.g. background colour. Another reason is that it is able to theme graphical objects using CSS under JavaFX, which makes the changes of styling properties of different components in the processor easier to code. Also, the Table View of JavaFX provide more functions like callbacks and the table.refresh() function is especially useful as there are frequent changes on main memory table, cache memory table and registers table throughout the instruction set execution. A final reason for using JavaFX is for the purpose of future development, although currently there are no animated effect added on the execution simulation, animation effects using JavaFX library are easier to code than other Java GUI library, thus in view of both current and future development of the Simulator, JavaFX library is adopted for implementing the GUI feature.

Threading

There are numbers of data movement and transform in one instruction execution, in order to show the changes of every step in an instruction execution on the screen, program sleep has to be introduced. However, the whole program is paused when a program sleep command is called, the GUI is also frozen and the changes on styling of components, e.g. highlighting PC label with yellow, are unable to display, therefore usage of threads are introduced.



As illustrated by the right flow diagram, there are several steps in each instruction execution, after running a step, the program will create a new thread, and the new thread will immediately sleep at start. In such way, the previous thread's changes on components on GUI will still take into effect, and the effect will stay on the screen until the new thread wake up and continue to run next step and cover the changes in previous thread (as GUI components are declared as global variable in Execution Class).

Project Results

Overview

There are 2 final output of this project, a GUI version simulator and command-line version simulator.

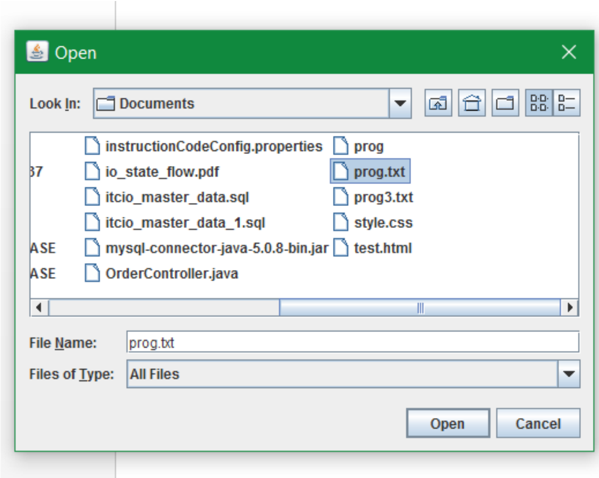
GUI version simulator serves as the deliverable that fully meet the major objective of this project as stated in the Introduction part, and it is regarded as the major product of this project.

While the command-line version, which is regarded as a “side-product”, can serve for several purposes. The first one is for logging use, as the output, which displaying the step one-by-one in sentence form, can be easily copied to a text file. It is still an improvement from one of the existing teaching aid (the C++ simulator program), as it covers Cache Memory in the simulation; furthermore for each instruction execution, content of register files and cache memory will also be shown on top of content of main memory, which is a step forward than the existing one. In addition, for future development of GUI version simulator, the command-line version can be used as an oracle to cross-check with the further developed simulator for testing and debugging.

Both version of the simulator will be a java executable file, thus access to source code can be restricted from general users (i.e. students).

Entry Window

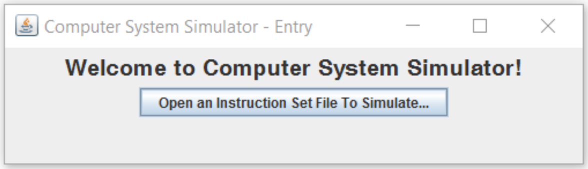
The right is the Simulator Entry Window containing a welcome message and File Chooser button. After clicking the File Chooser button, the left File Chooser Dialogue will be popped up and let user choose the instruction set binary text file from the directory system.



Look In: Documents

File Name: prog.txt

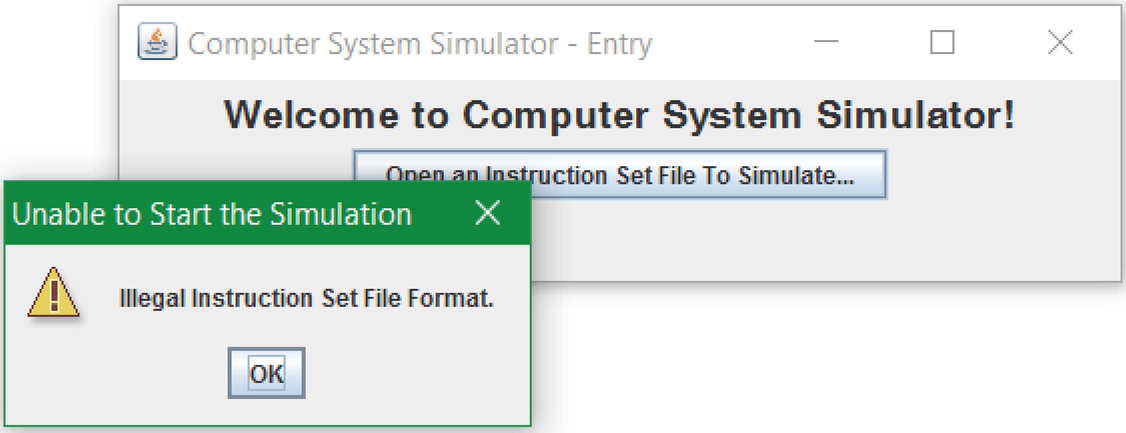
Files of Type: All Files


Computer System Simulator - Entry
Open an Instruction Set File To Simulate...

Welcome to Computer System Simulator!

Illegal File Type Check

The Simulator will only accept plain text file (file extension of .txt or pure binary file) as legal instruction set file format, the Simulator will not proceed and prompt a warning message to ask user to choose other input file if it detects input file format other than the above mentioned file type.


Computer System Simulator - Entry
Open an Instruction Set File To Simulate...
OK

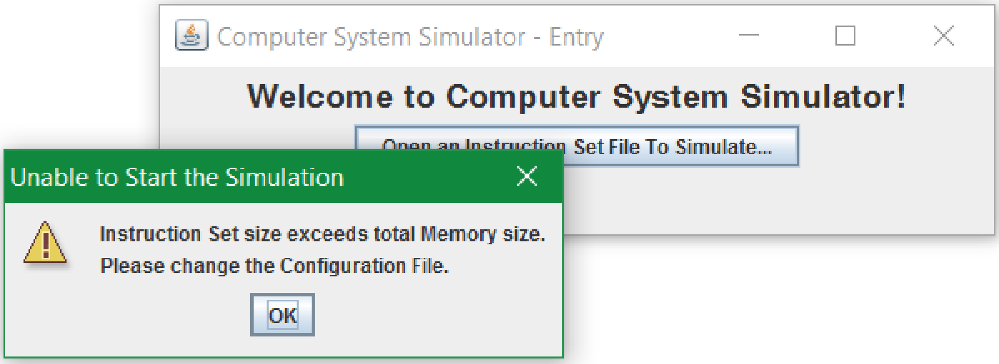
Welcome to Computer System Simulator!

Unable to Start the Simulation

Illegal Instruction Set File Format.

Check Instruction Set Size against Configuration

The Simulator will check whether the size of instruction set is within the main memory limit stated in the external configuration file, if it exceeds the Simulator will not proceed and prompt a warning message to ask user to modify the configuration file.


Computer System Simulator - Entry
Open an Instruction Set File To Simulate...
OK

Welcome to Computer System Simulator!

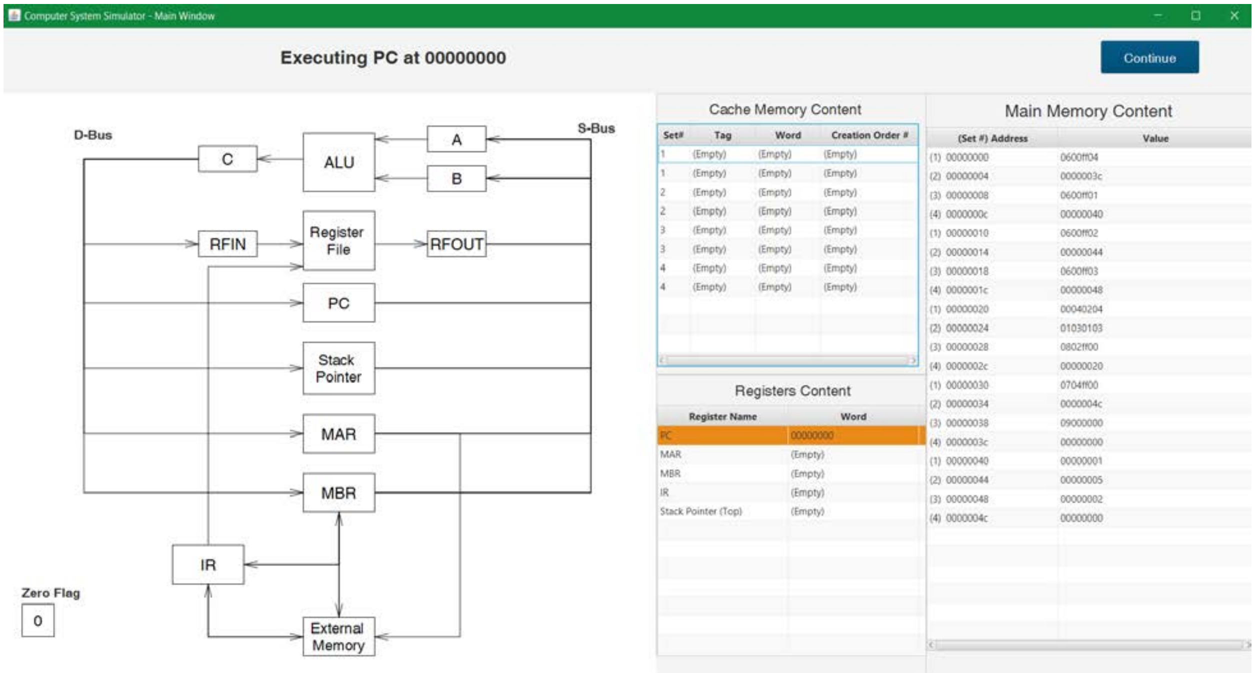
Unable to Start the Simulation

Instruction Set size exceeds total Memory size.
Please change the Configuration File.

Main Window (Execution)

Base (Idle) State

A new window for demonstrating the Execution will be displayed if the instruction set input passed the 2 validation in the Entry window.



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

Indicating Changes

The way to indicate changes occurred are slightly different for different part on the Execution Window, thus each part in the Main Window is "symbolized" by different colour, making users easier to notice which parts in the whole large Window are involved in the change.

For the left canvas, involved components will be highlighted with yellow background. For some scenario the content of the register will also be displayed on the component label so as to illustrate the data movement or transformation more clearly, e.g. for the step when ALU perform ADD, register A, B and C will display its content such as "00000011", so that it will be easier for student to get that C stores the value from addition of content in register A 7 register B. Contents of PC, MAR, MBR, Stack Pointer and IR will also be shown in the "Registers Content" table in the lower center pane.

For buses involved in the execution step, they will be highlighted in orange and the line weight will be slightly increased.

For Cache Memory Content table, the affected row will be highlighted in light-blue.

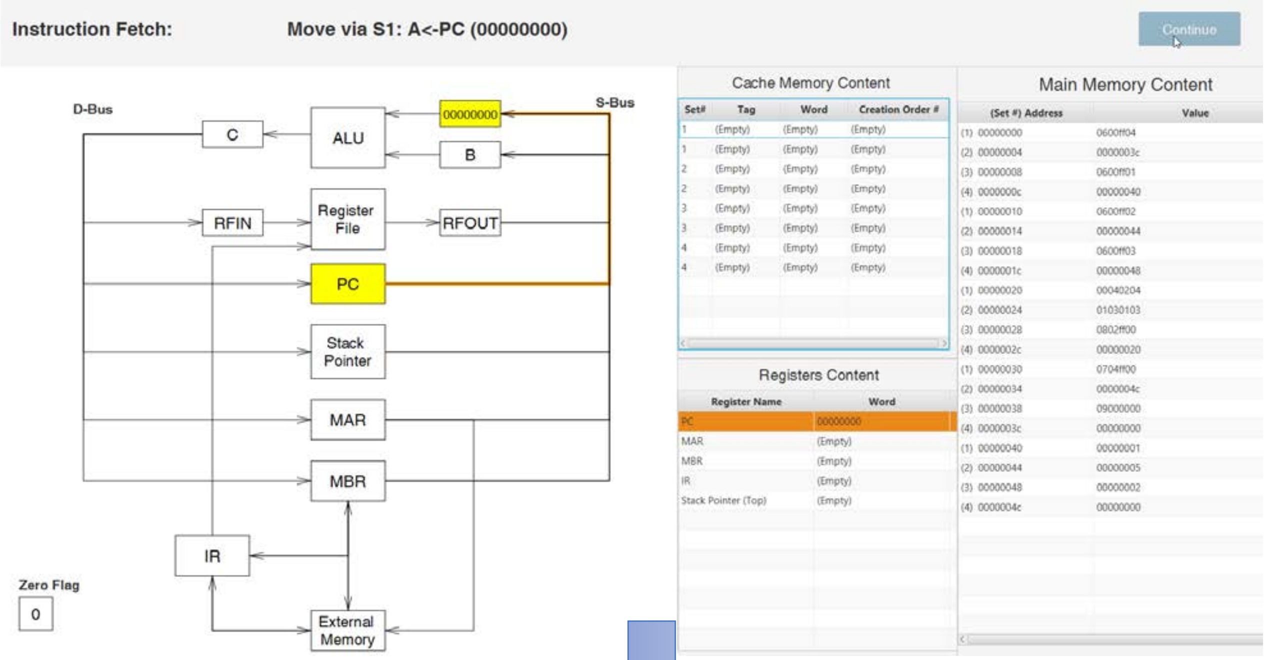
For Registers Content table, the rows involved will be highlighted in orange.

For Main Memory Content table, the row involved will be highlighted in light green.

Instruction Fetch

Example Screen Captures to illustrate the changes on the simulator in Instruction Fetch:

The leftmost part on top pane stated that the current step is in "Instruction Fetch" process



Instruction Fetch: **Move via S1: A<-PC (00000000)**

Cache Memory Content

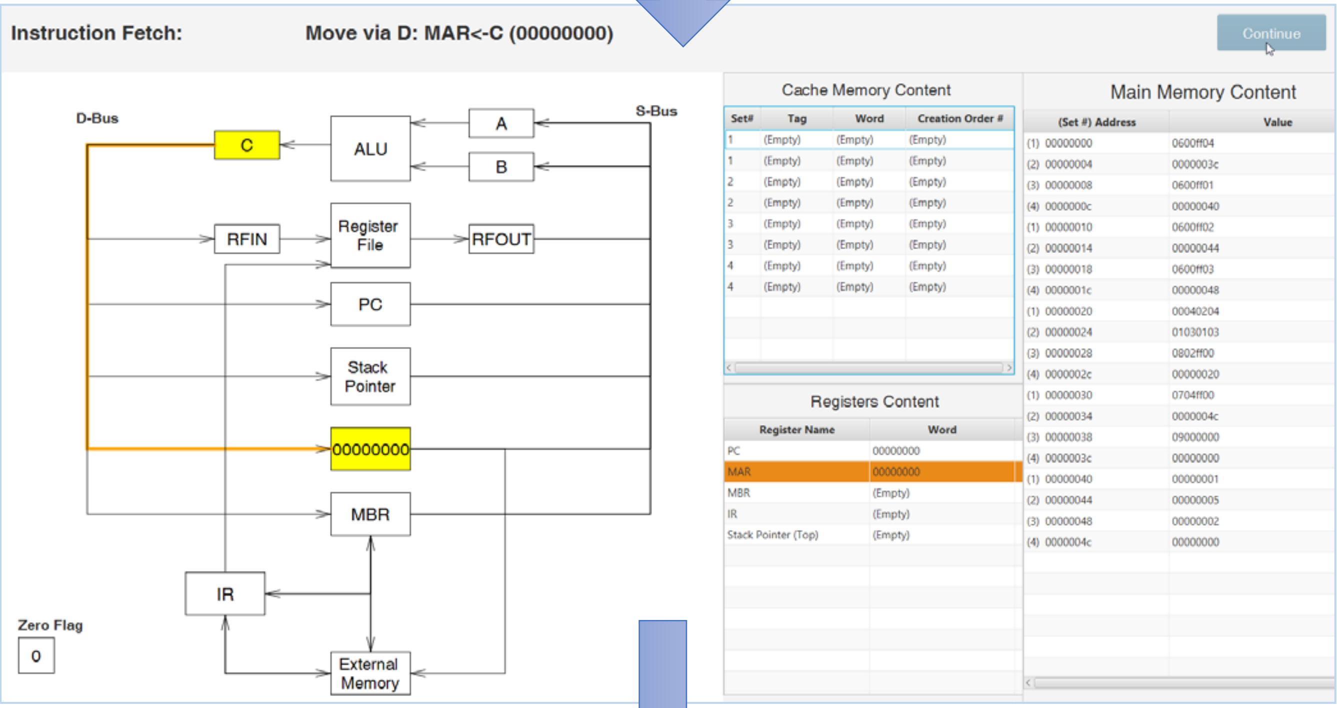
Set#	Tag	Word	Creation Order #
1	(Empty)	(Empty)	(Empty)
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

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



Instruction Fetch: **Move via D: MAR<-C (00000000)**

Cache Memory Content

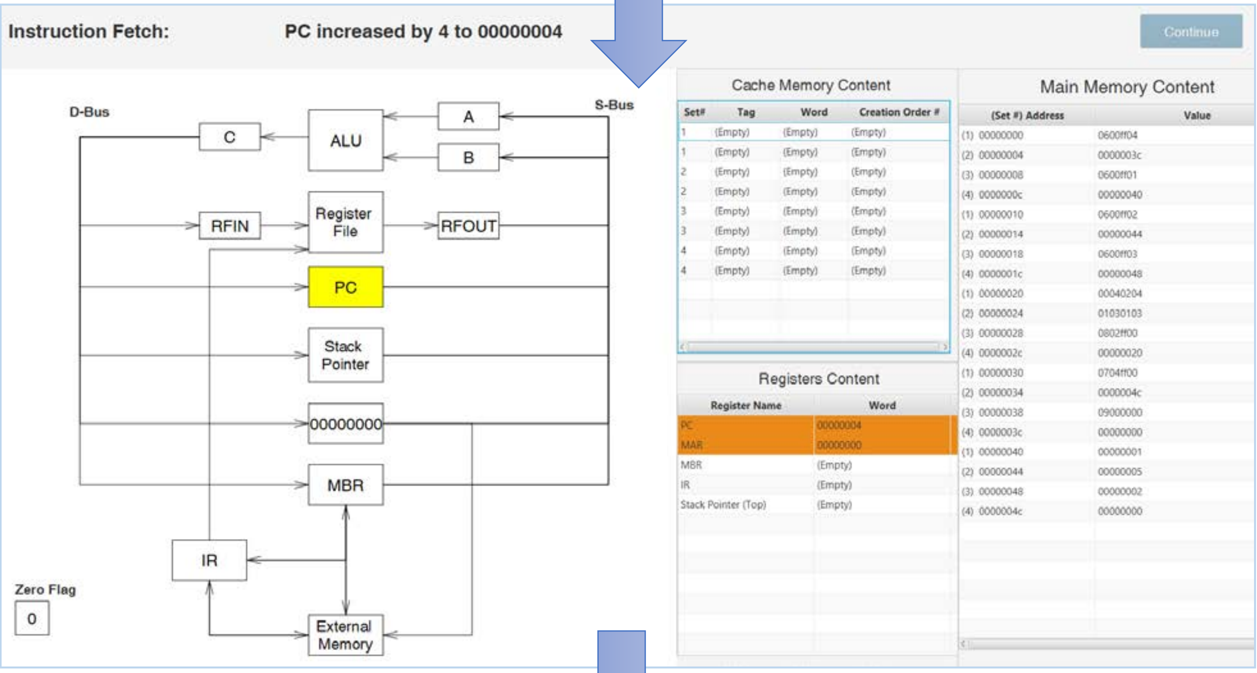
Set#	Tag	Word	Creation Order #
1	(Empty)	(Empty)	(Empty)
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000000
MAR	00000000
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



Instruction Fetch: PC increased by 4 to 00000004

Cache Memory Content

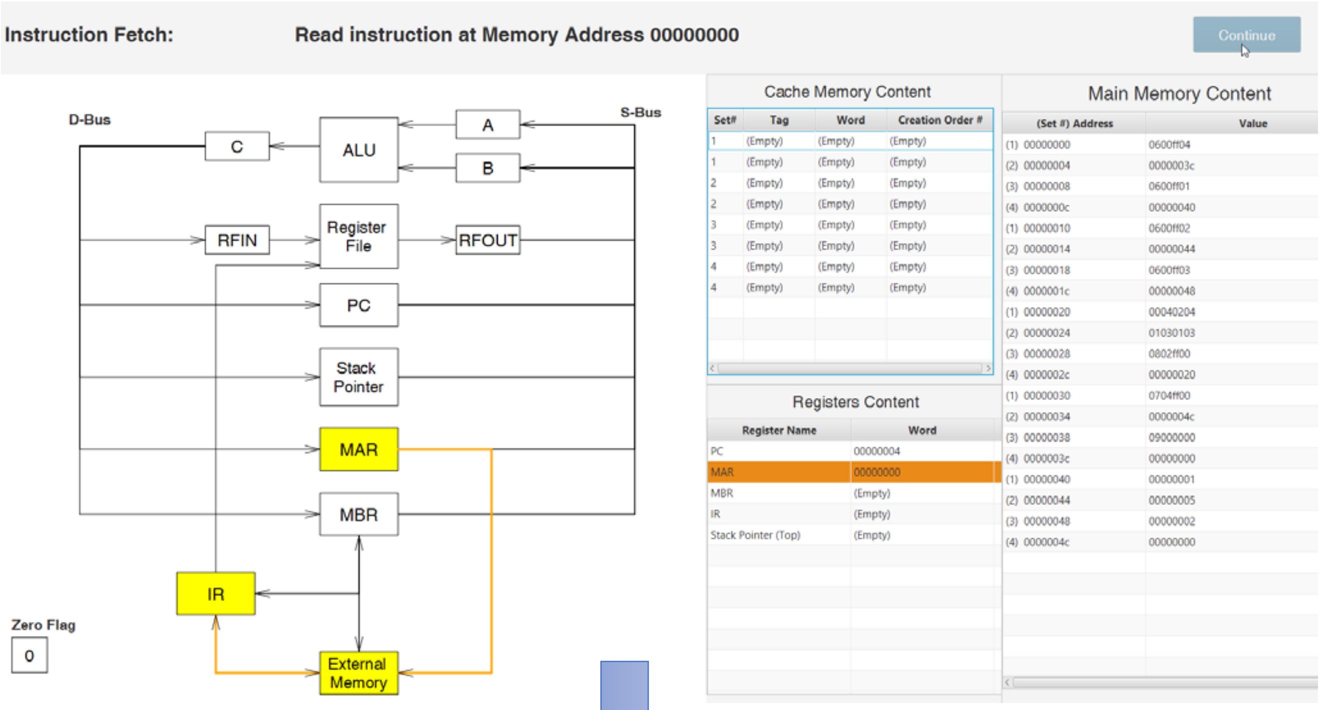
Set#	Tag	Word	Creation Order #
1	(Empty)	(Empty)	(Empty)
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000004
MAR	00000000
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



Instruction Fetch: Read instruction at Memory Address 00000000

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)

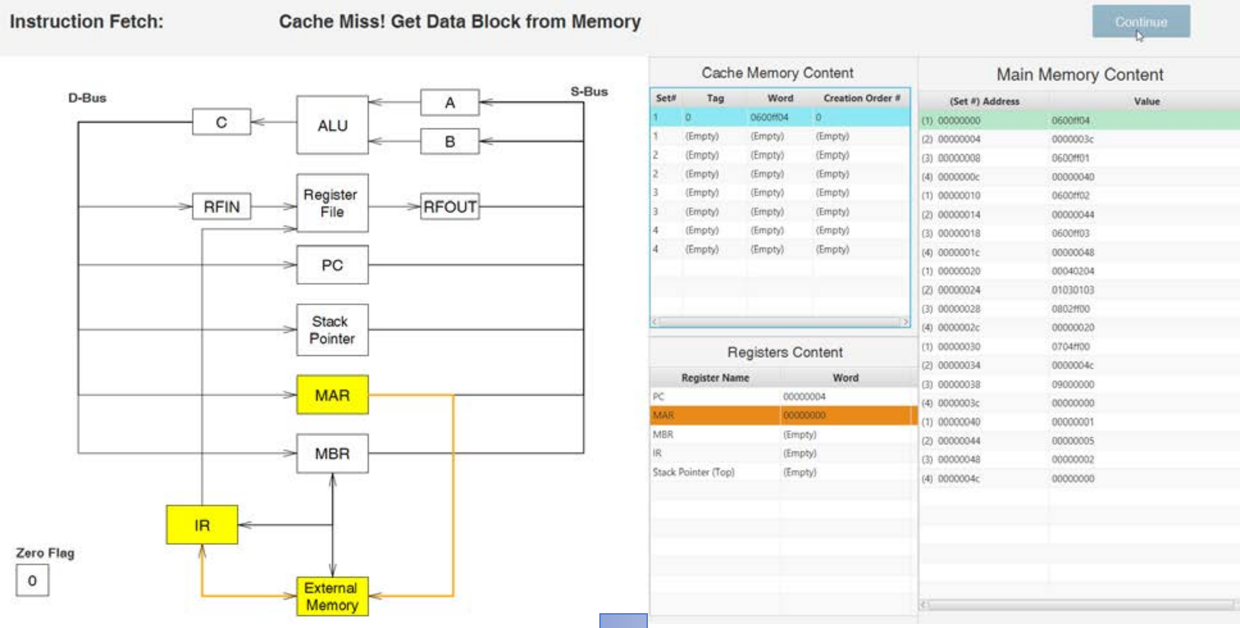
Registers Content

Register Name	Word
PC	00000004
MAR	00000000
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

**It is possible to have Cache Hit in this Step (e.g. when there is Branch operation) **



Instruction Fetch: **Cache Miss! Get Data Block from Memory**

Cache Memory Content

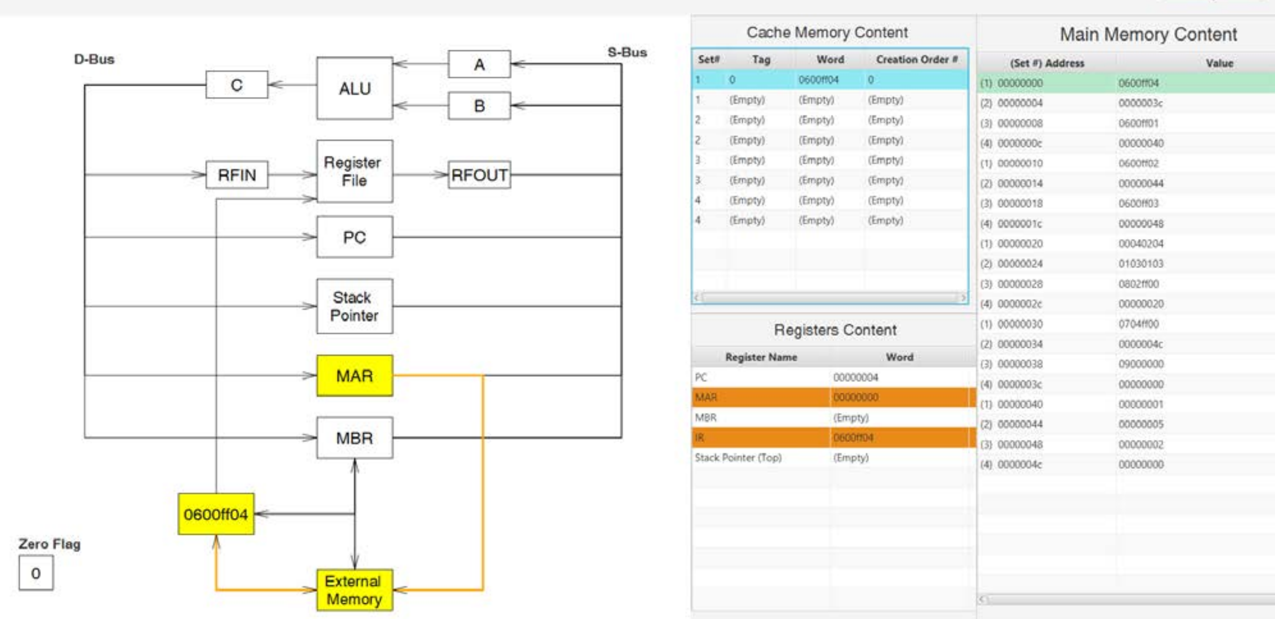
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000004
MAR	00000000
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



Instruction Fetch: **IR <- 0600ff04**

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
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	00000004
MAR	00000000
MBR	(Empty)
IR	0600ff04
Stack Pointer (Top)	(Empty)

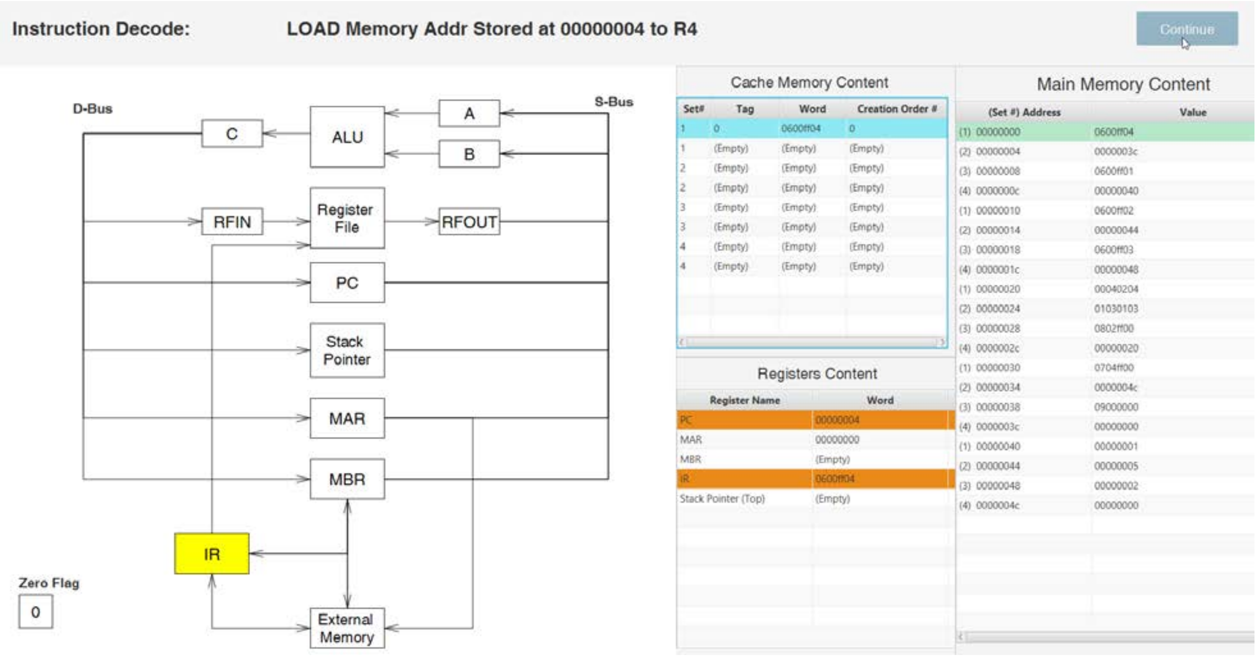
Main Memory Content

(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

Instruction Decode

In this step, the instruction is decoded and displayed as human-readable description at the top pane. Meanwhile, the leftmost part of the top pane has been changed from "Instruction Fetch" to "Instruction Decode".

Example Screen Capture for decoding LOAD instruction:



Instruction Decode: **LOAD Memory Addr Stored at 00000004 to R4**

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
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)

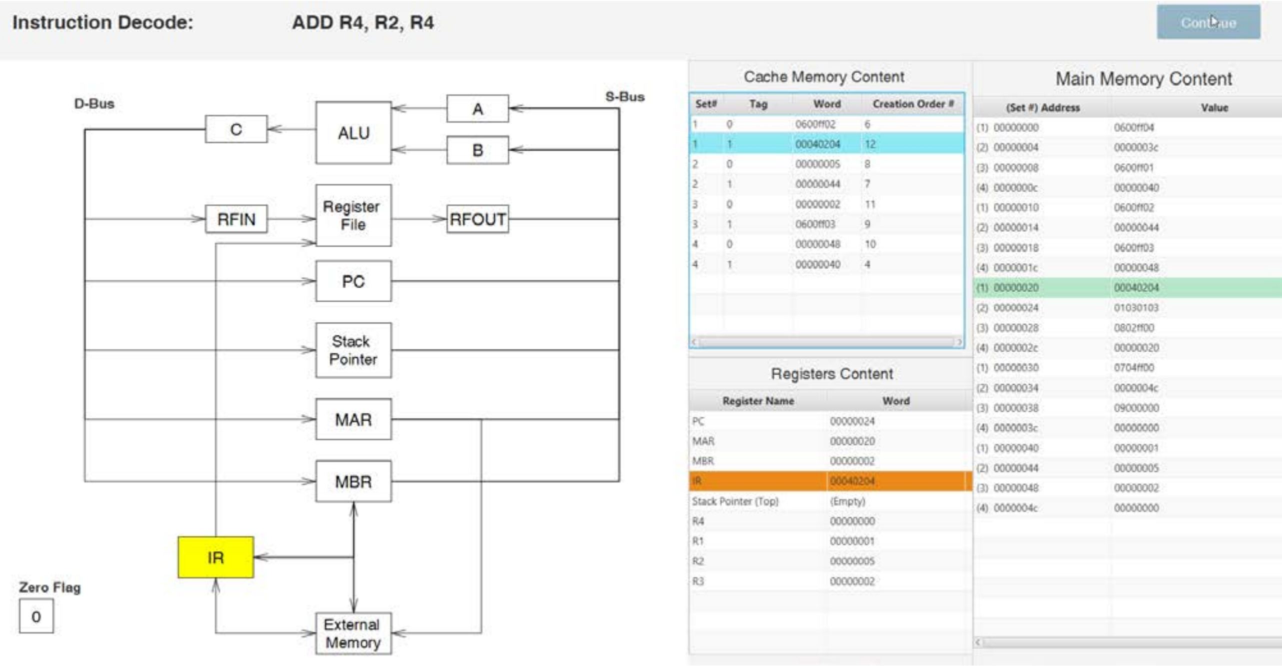
Registers Content

Register Name	Word
PC	00000004
MAR	00000000
MBR	(Empty)
IR	0600ff04
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

Example Screen Capture for decoding ADD instruction:



Instruction Decode: **ADD R4, R2, R4**

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000000
R1	00000001
R2	00000005
R3	00000002

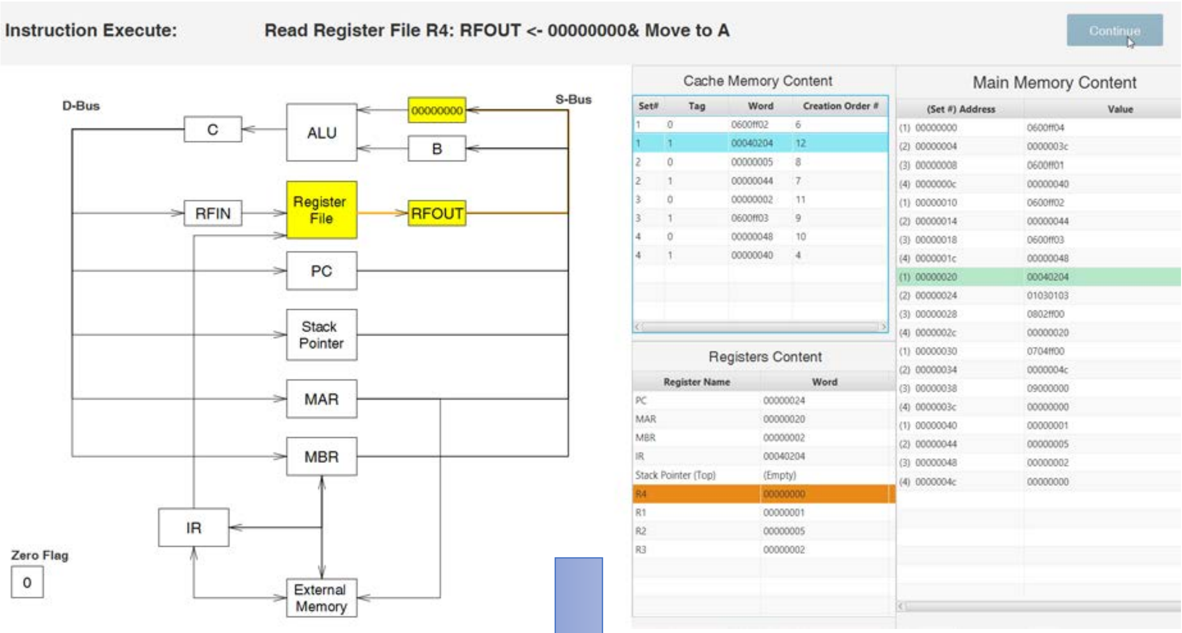
Main Memory Content

(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

Instruction Execution

The number of steps in Instruction Exception stage can varies greatly, for some operation such as LOAD, STORE and BRANCH, there are more number of steps as the processor has to retrieve content of the next memory address as well in the operation. To reduce number of frame changing, some steps will be slightly simplified, which would be illustrated in the example screen captures below.

Example Screen Captures for Instruction Execution stage of an instruction that does not involve accessing next memory, e.g. ADD, SUB, MOV:



Instruction Execute: **Read Register File R4: RFOUT <- 00000000& Move to A**

Cache Memory Content

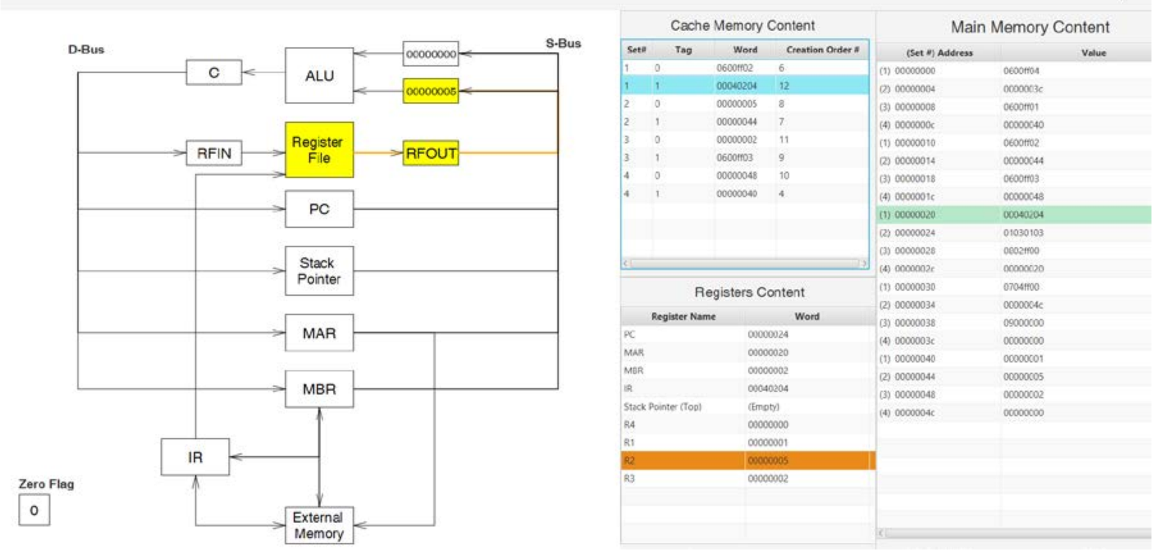
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000000
R1	00000001
R2	00000005
R3	00000002

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



Instruction Execute: **Read Register File R2: RFOUT <- 00000005& Move to B**

Cache Memory Content

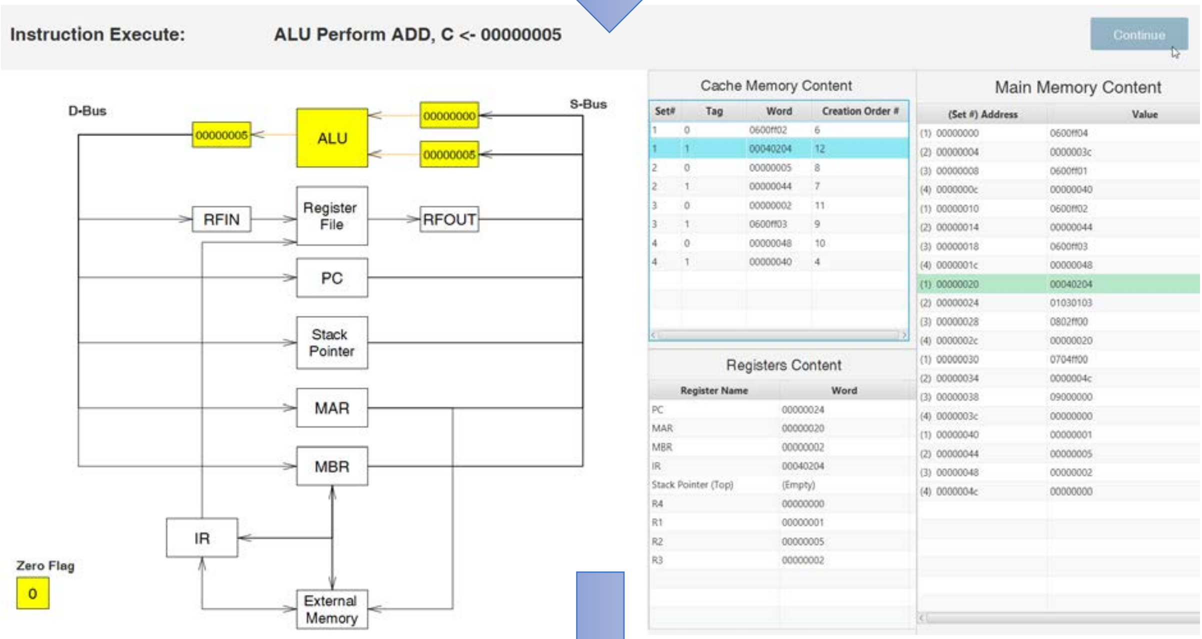
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000000
R1	00000001
R2	00000005
R3	00000002

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



Instruction Execute: ALU Perform ADD, C <- 00000005

Cache Memory Content

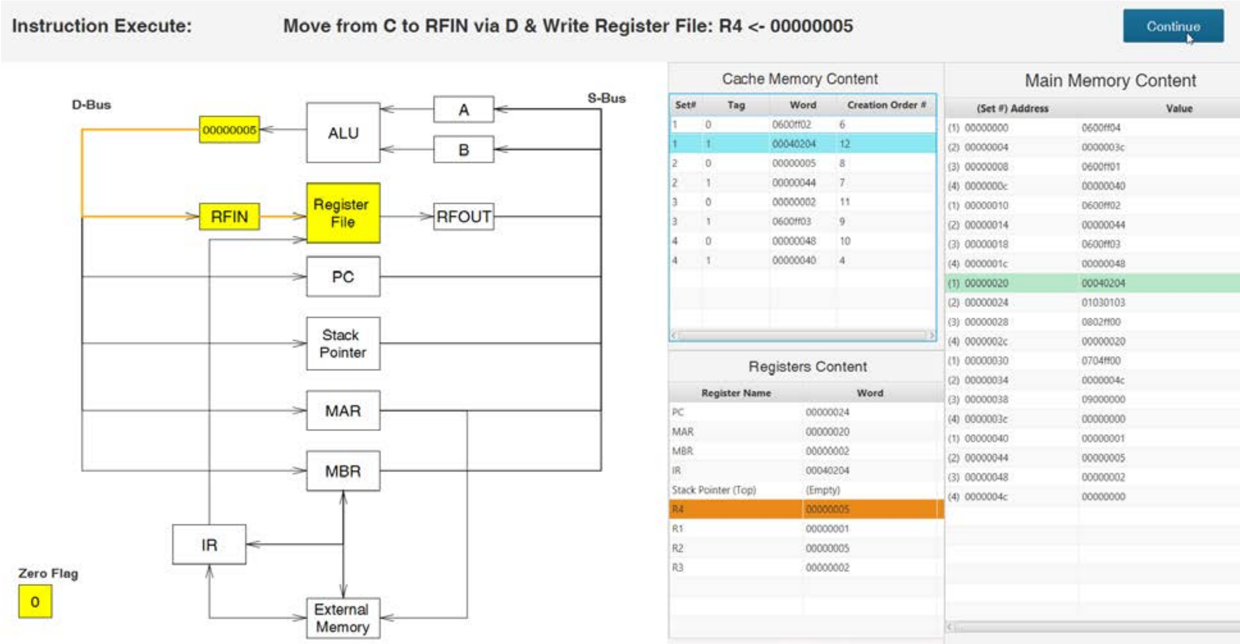
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000000
R1	00000001
R2	00000005
R3	00000002

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



Instruction Execute: Move from C to RFIN via D & Write Register File: R4 <- 00000005

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

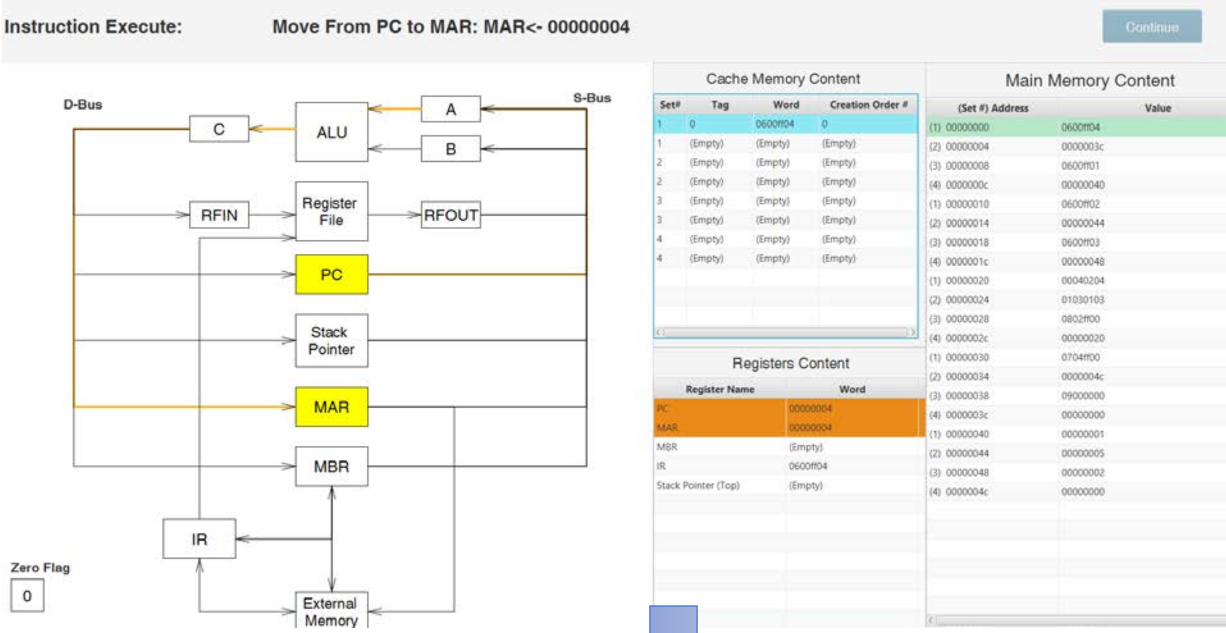
Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000002

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

Example Screen Captures for Instruction Execution stage of an instruction that involves accessing next memory, e.g. LOAD, STORE:



Instruction Execute: **Move From PC to MAR: MAR<- 00000004**

Cache Memory Content

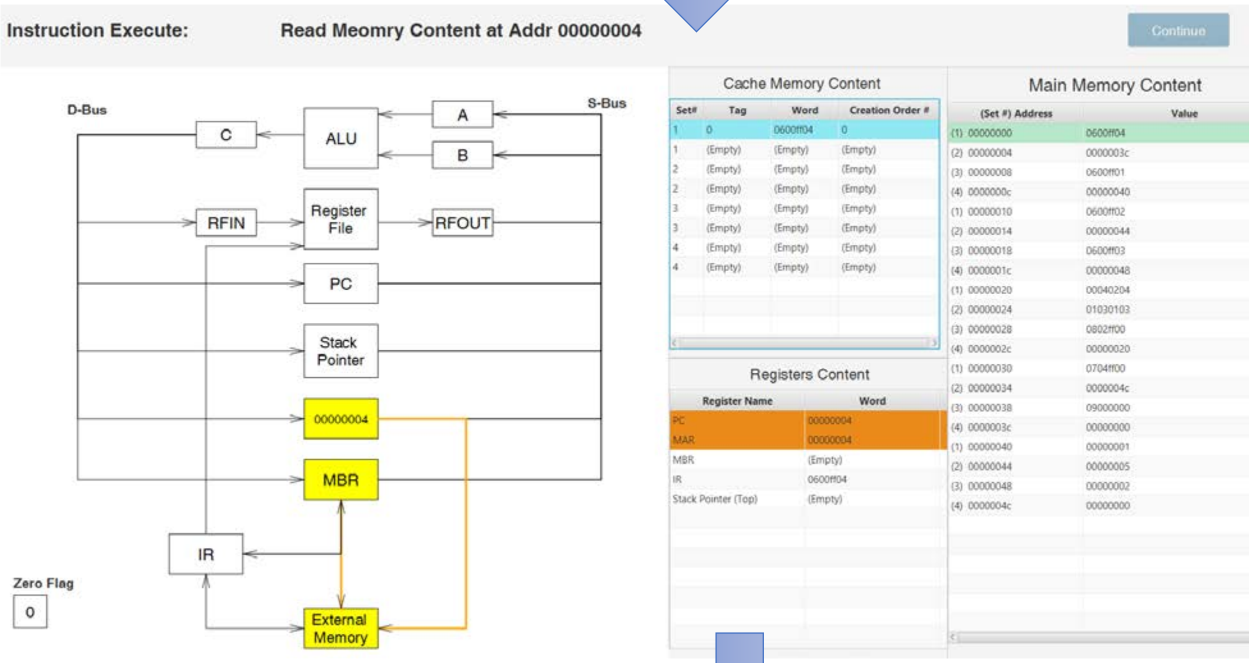
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000004
MAR	00000004
MBR	(Empty)
IR	0600ff04
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



Instruction Execute: **Read Meomry Content at Addr 00000004**

Cache Memory Content

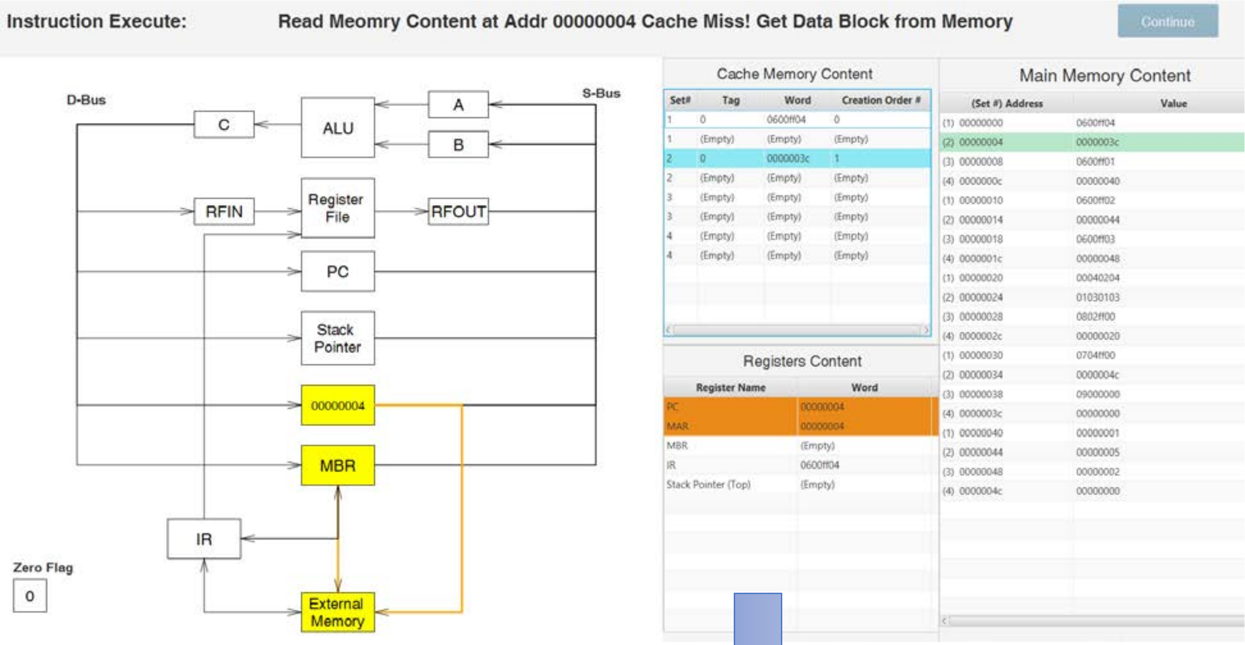
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000004
MAR	00000004
MBR	(Empty)
IR	0600ff04
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



Instruction Execute: Read Meomry Content at Addr 00000004 Cache Miss! Get Data Block from Memory

Cache Memory Content

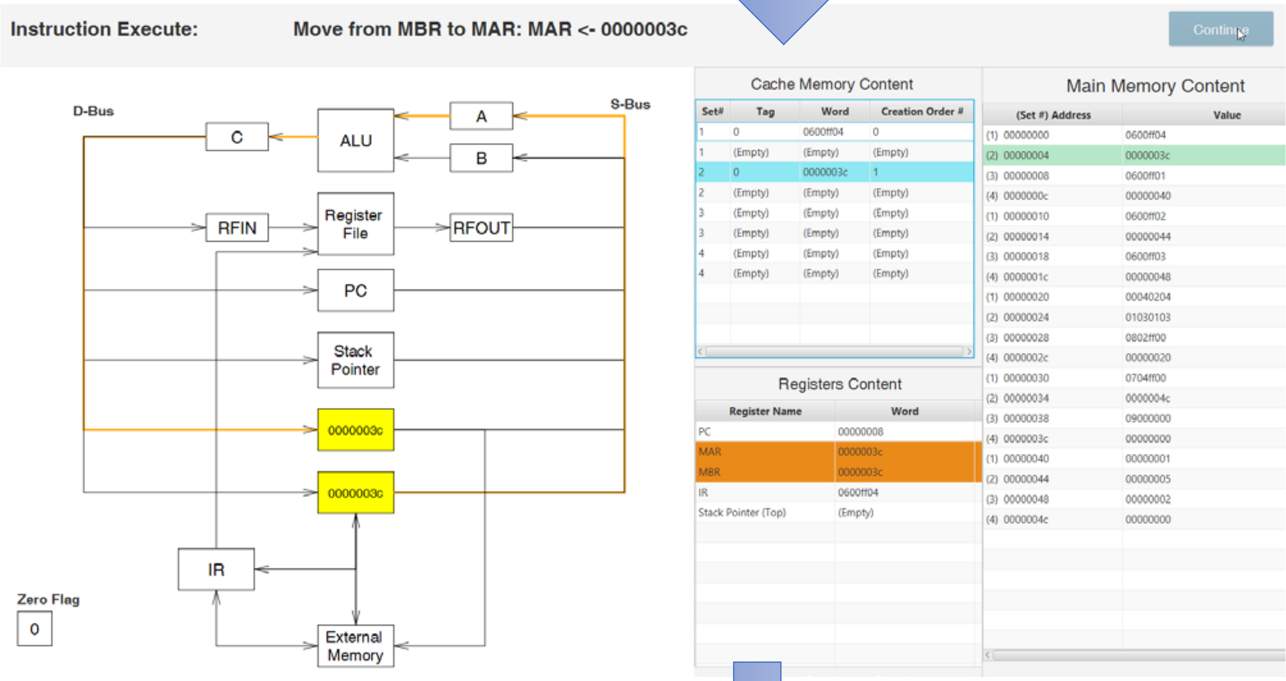
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000004
MAR	00000004
MBR	(Empty)
IR	0600ff04
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



Instruction Execute: Move from MBR to MAR: MAR <- 0000003c

Cache Memory Content

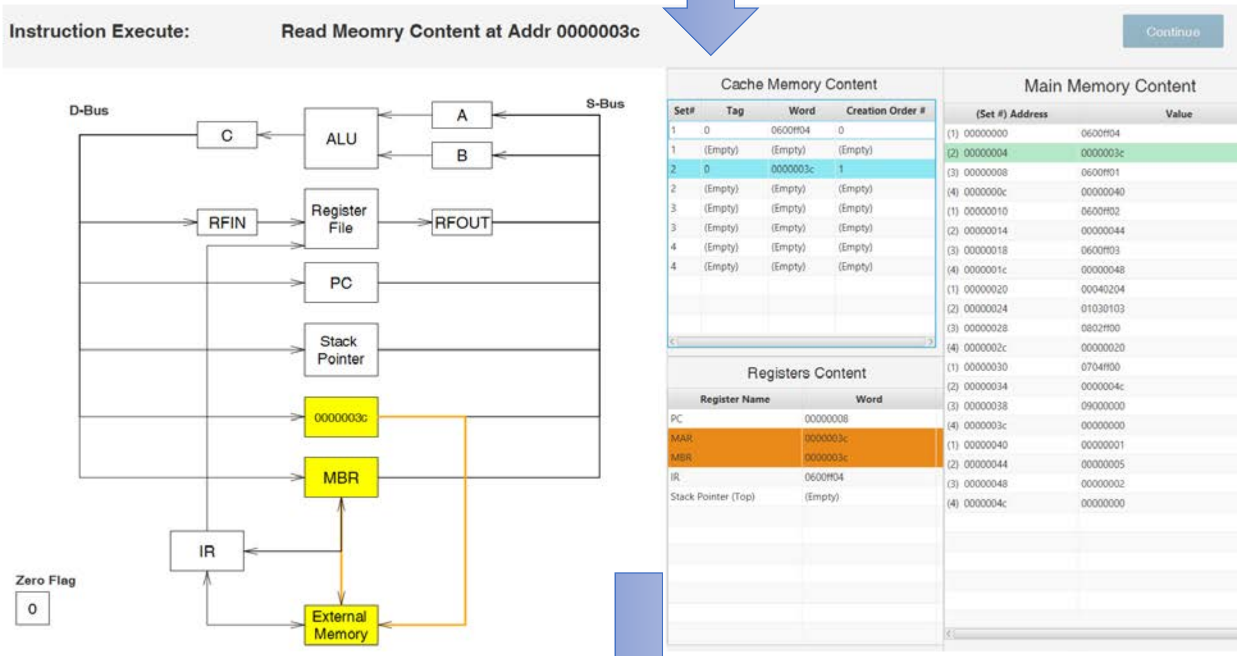
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000008
MAR	0000003c
MBR	0000003c
IR	0600ff04
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



Instruction Execute: **Read Meomry Content at Addr 0000003c**

Cache Memory Content

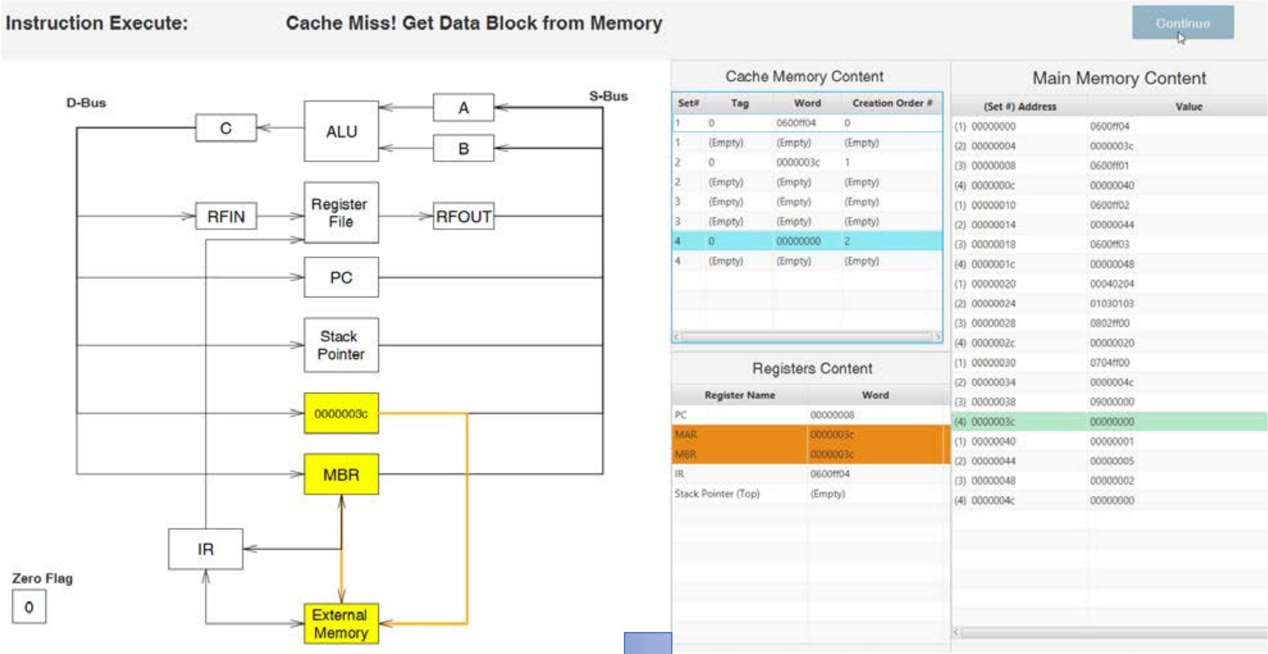
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000008
MAR	0000003c
MBR	0000003c
IR	0600ff04
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



Instruction Execute: **Cache Miss! Get Data Block from Memory**

Cache Memory Content

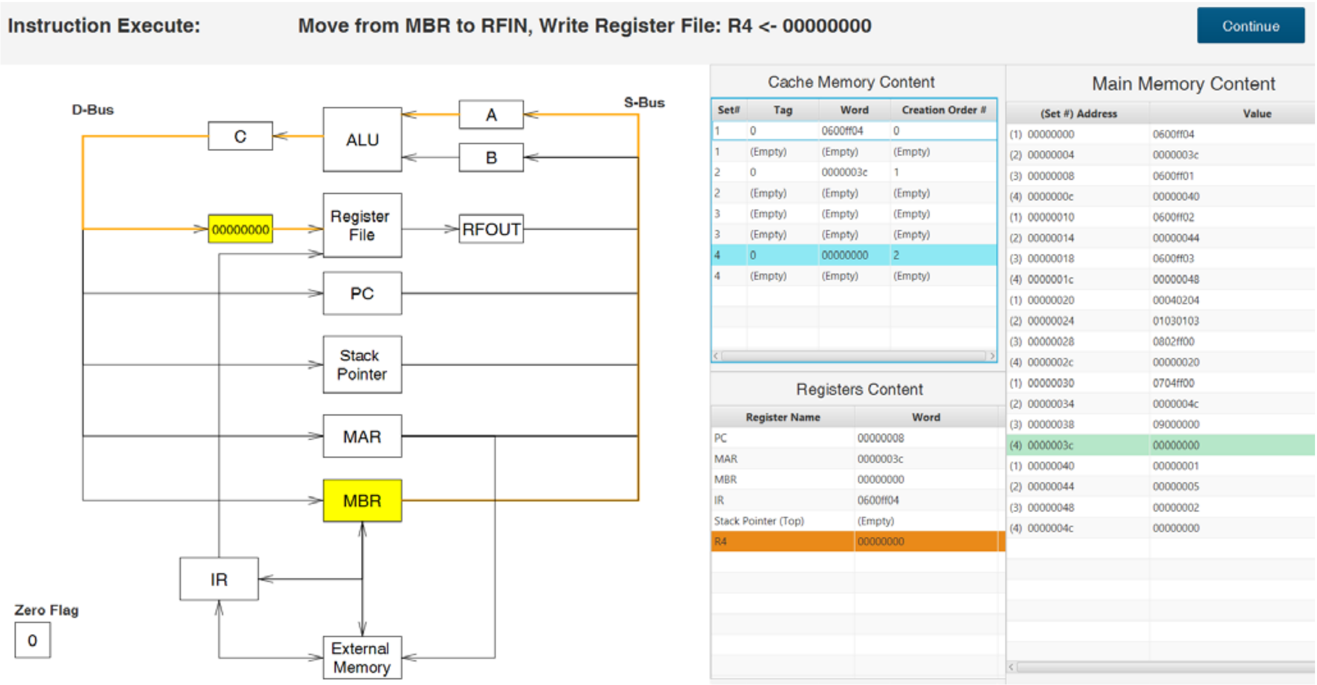
Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	0	00000000	2
4	(Empty)	(Empty)	(Empty)

Registers Content

Register Name	Word
PC	00000008
MAR	0000003c
MBR	0000003c
IR	0600ff04
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



Instruction Execute: Move from MBR to RFIN, Write Register File: R4 <- 00000000

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
3	(Empty)	(Empty)	(Empty)
4	0	00000000	2
4	(Empty)	(Empty)	(Empty)

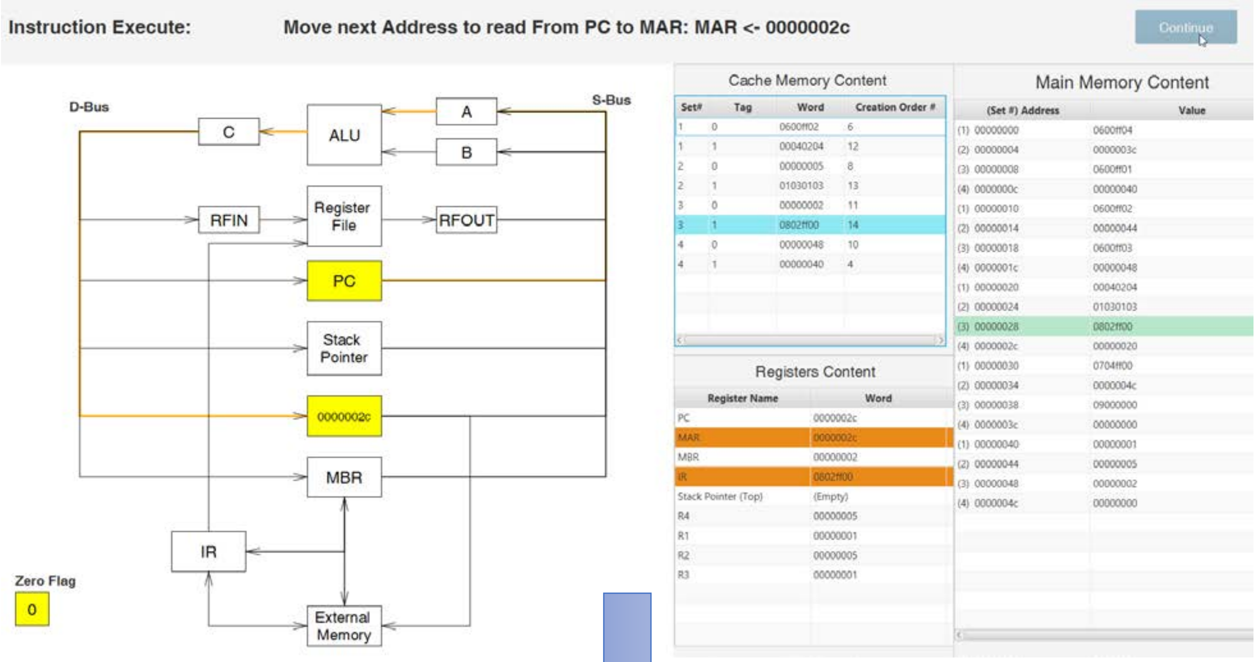
Registers Content

Register Name	Word
PC	00000008
MAR	0000003c
MBR	00000000
IR	0600ff04
Stack Pointer (Top)	(Empty)
R4	00000000

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

Example Screen Captures for Instruction Execution stage of a BRANCH instruction:



Instruction Execute: Move next Address to read From PC to MAR: MAR <- 0000002c

Cache Memory Content

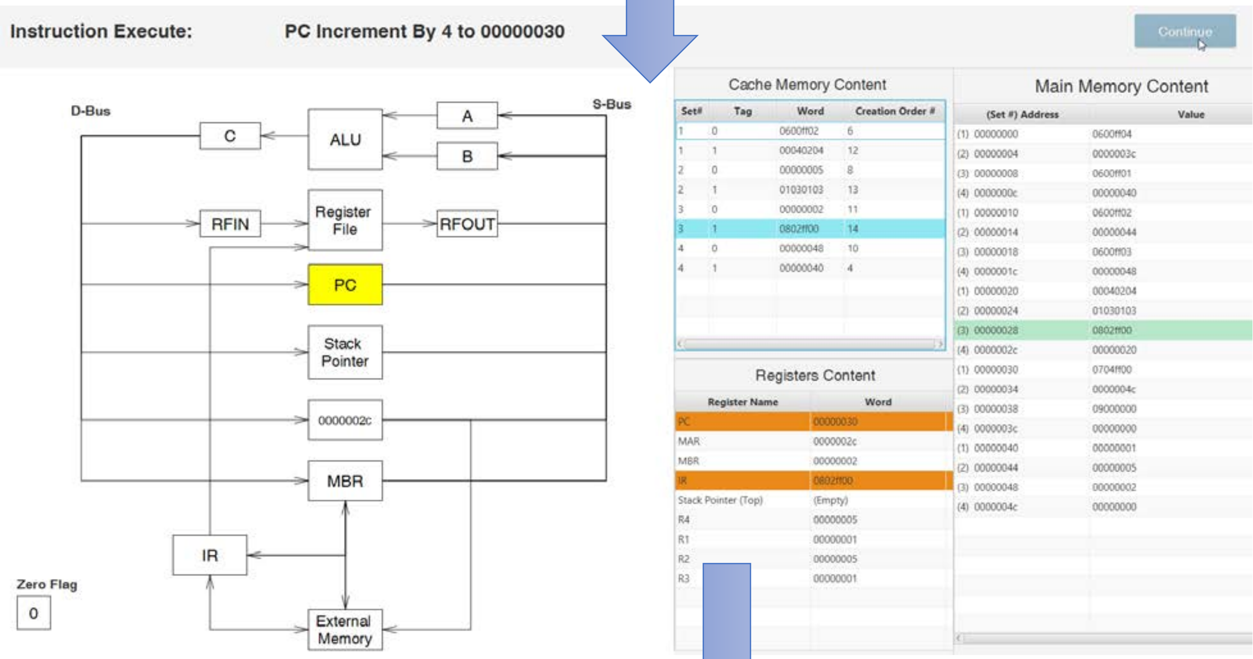
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	0000002c
MAR	0000002c
MBR	00000002
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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



Instruction Execute: PC Increment By 4 to 00000030

Cache Memory Content

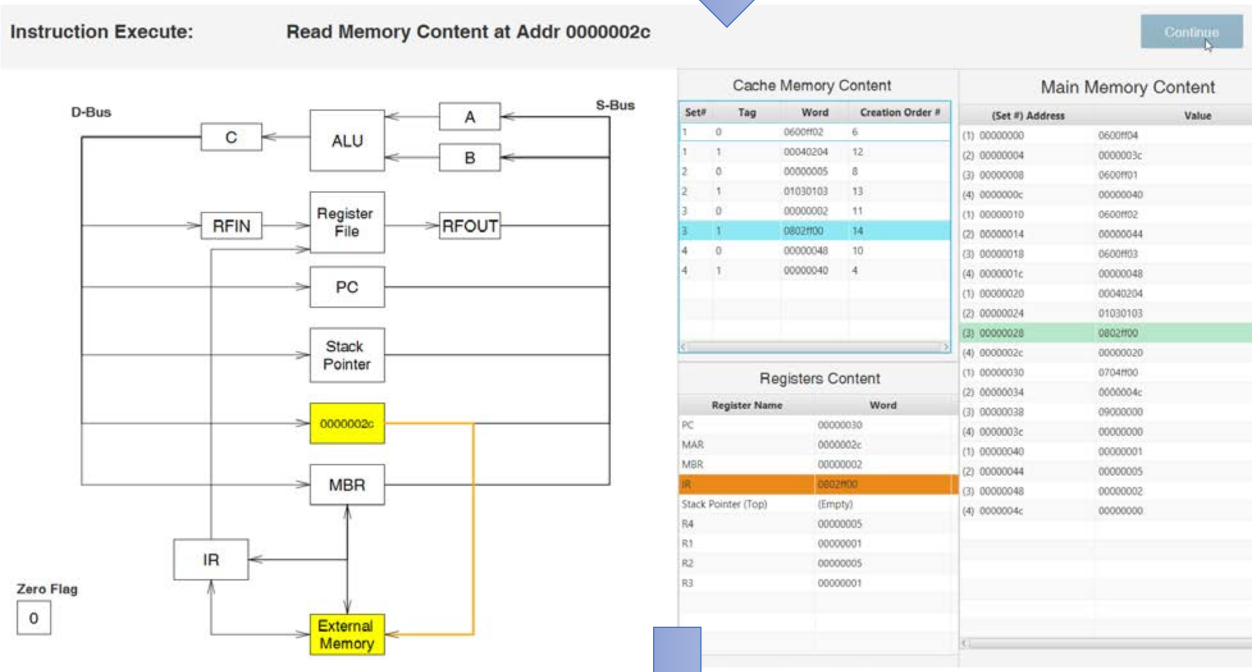
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000030
MAR	0000002c
MBR	00000002
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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



Instruction Execute: Read Memory Content at Addr 0000002c

Cache Memory Content

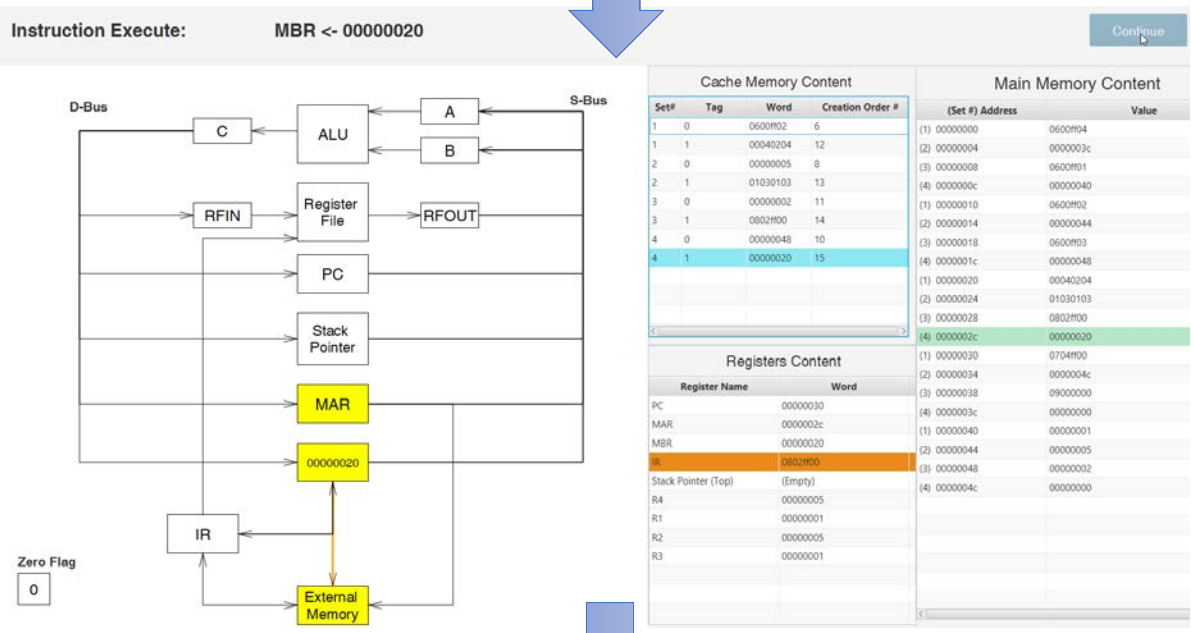
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000030
MAR	0000002c
MBR	00000002
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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



Instruction Execute: MBR <- 00000020

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

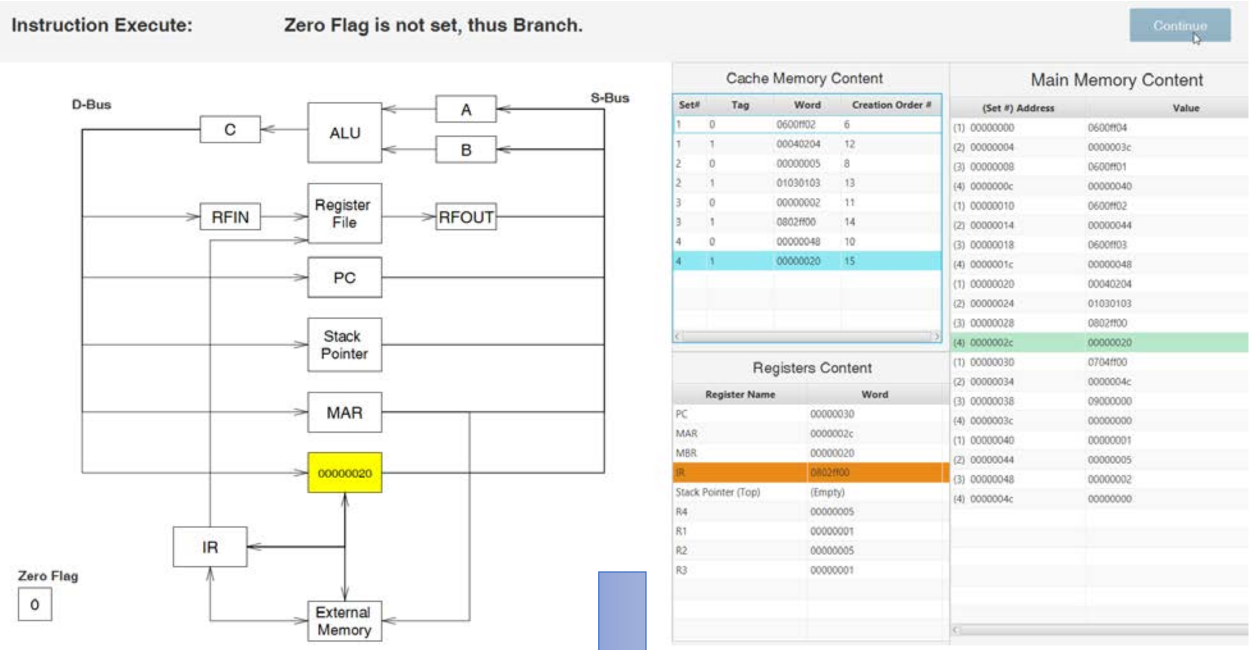
Registers Content

Register Name	Word
PC	00000030
MAR	0000002c
MBR	00000020
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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

**Scenario 1: BNZ (Branch If Not Zero) operation & Zero Flag is not set **



Instruction Execute: Zero Flag is not set, thus Branch.

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

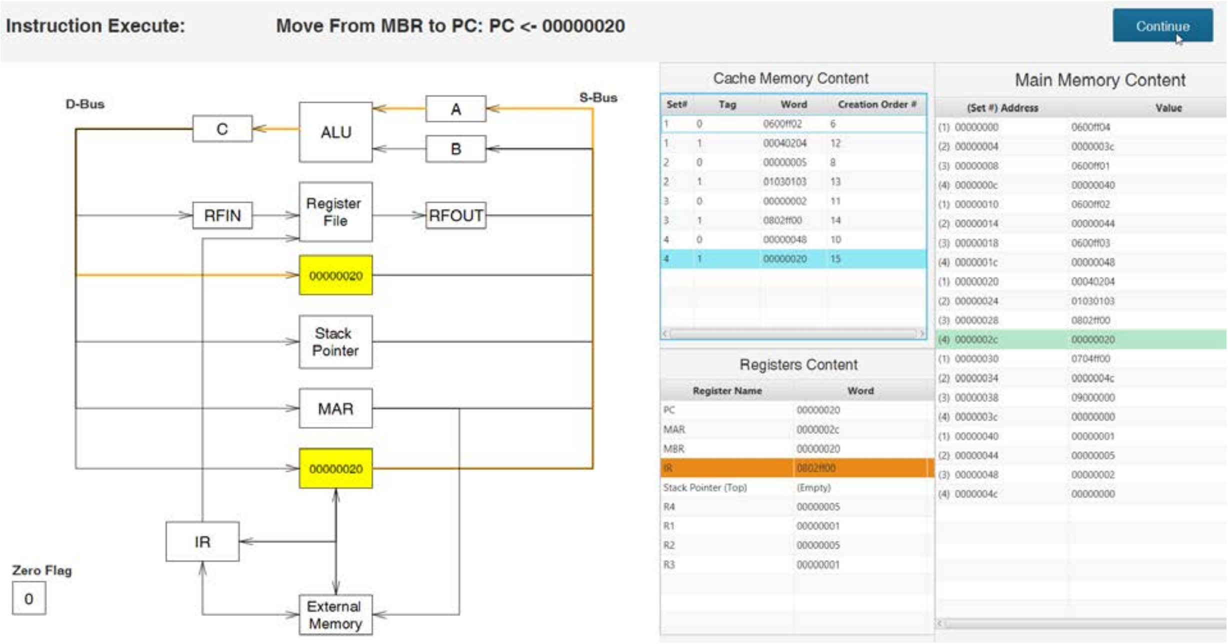
Registers Content

Register Name	Word
PC	00000030
MAR	0000002c
MBR	00000020
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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

**Scenario 1: BNZ operation & Zero Flag is not set ** (con't)



Instruction Execute: Move From MBR to PC: PC <- 00000020

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

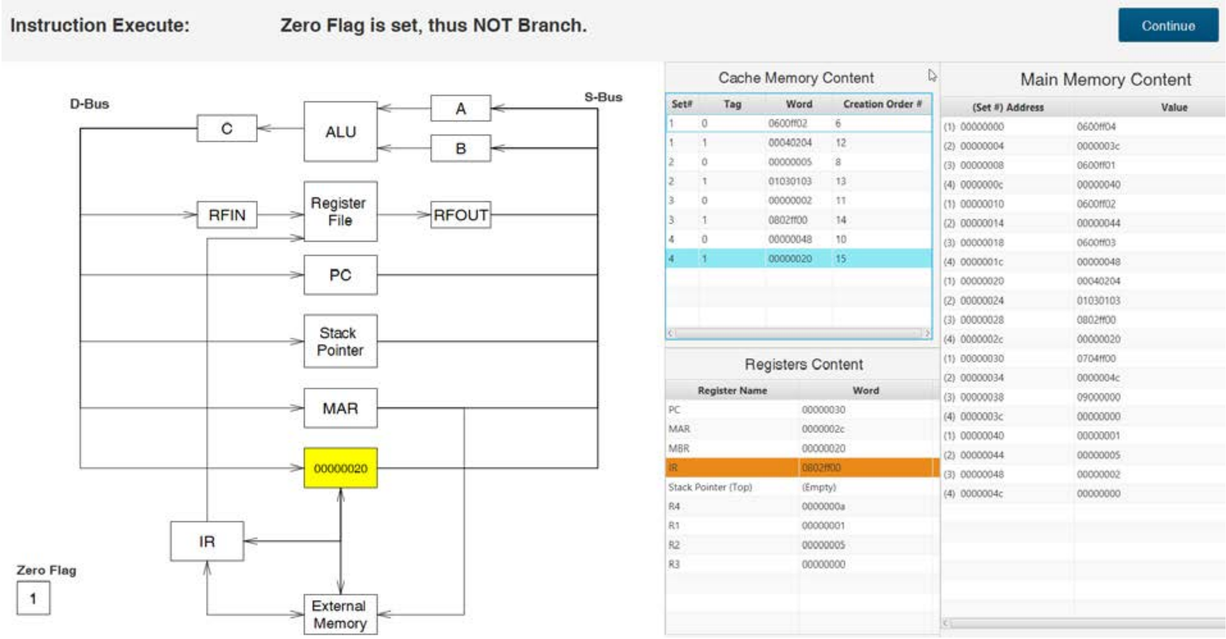
Registers Content

Register Name	Word
PC	00000020
MAR	0000002c
MBR	00000020
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	00000005
R1	00000001
R2	00000005
R3	00000001

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

**Scenario 2: BNZ operation & Zero Flag is set **



Instruction Execute: Zero Flag is set, thus NOT Branch.

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

Registers Content

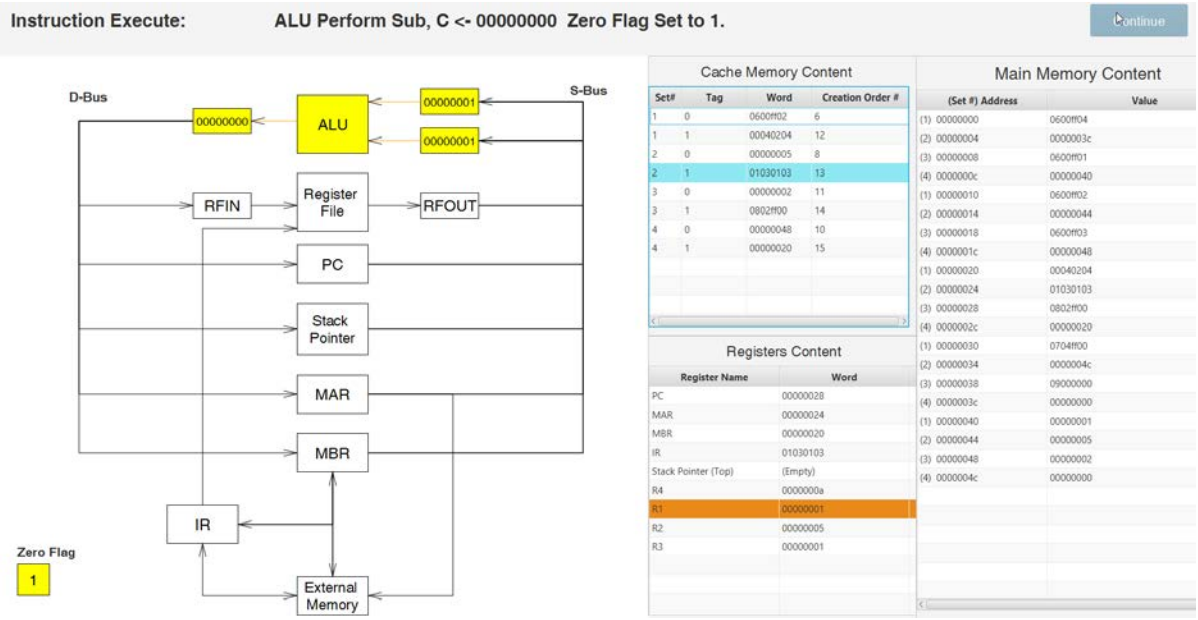
Register Name	Word
PC	00000030
MAR	0000002c
MBR	00000020
IR	0802ff00
Stack Pointer (Top)	(Empty)
R4	0000000a
R1	00000001
R2	00000005
R3	00000000

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

Zero Flag

Branch operations check whether zero flag is set to decide whether to branch or not. Zero Flag will be set when the operation in ALU is equal to zero. In the screen capture below, which is a subtraction instruction, as the result from ALU operation is 0, the zero flag is set to 1, and the zero flag label is highlighted in yellow (at bottom left corner).



Instruction Execute: ALU Perform Sub, C <- 00000000 Zero Flag Set to 1.

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

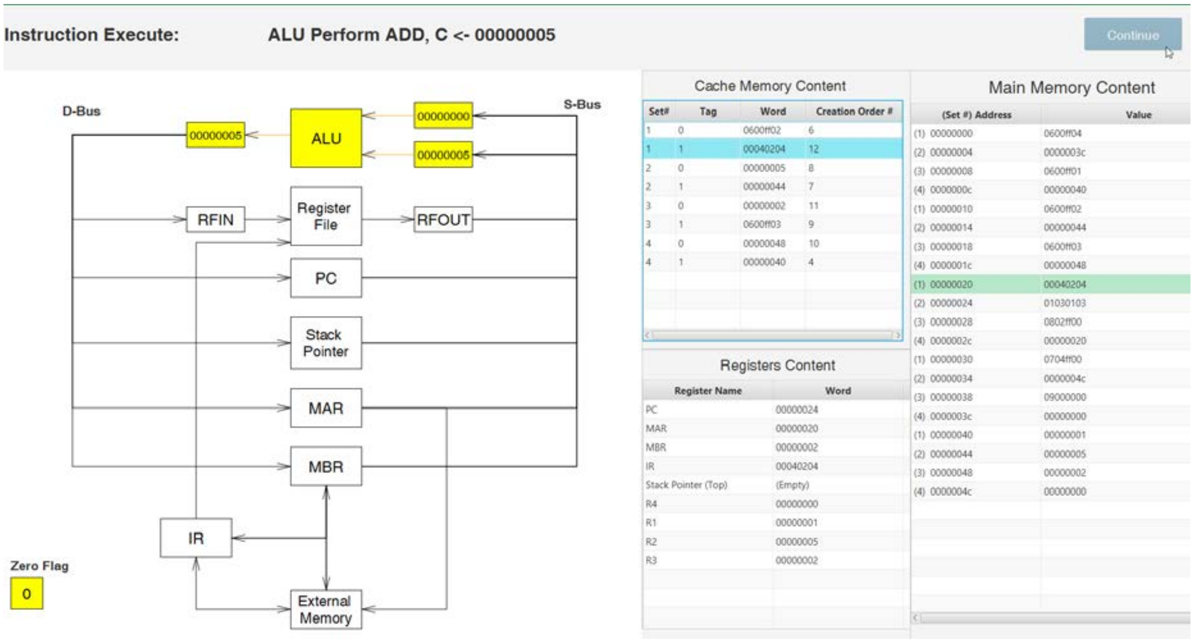
Registers Content

Register Name	Word
PC	00000028
MAR	00000024
MBR	00000020
IR	01030103
Stack Pointer (Top)	(Empty)
R4	0000000a
R1	00000001
R2	00000005
R3	00000001

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

When the result is not zero, like the Add operation below, the zero flag will be labelled as zero, indicate that zero flag is not set.



Instruction Execute: ALU Perform ADD, C <- 00000005

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	00000044	7
3	0	00000002	11
3	1	0600ff03	9
4	0	00000048	10
4	1	00000040	4

Registers Content

Register Name	Word
PC	00000024
MAR	00000020
MBR	00000002
IR	00040204
Stack Pointer (Top)	(Empty)
R4	00000000
R1	00000001
R2	00000005
R3	00000002

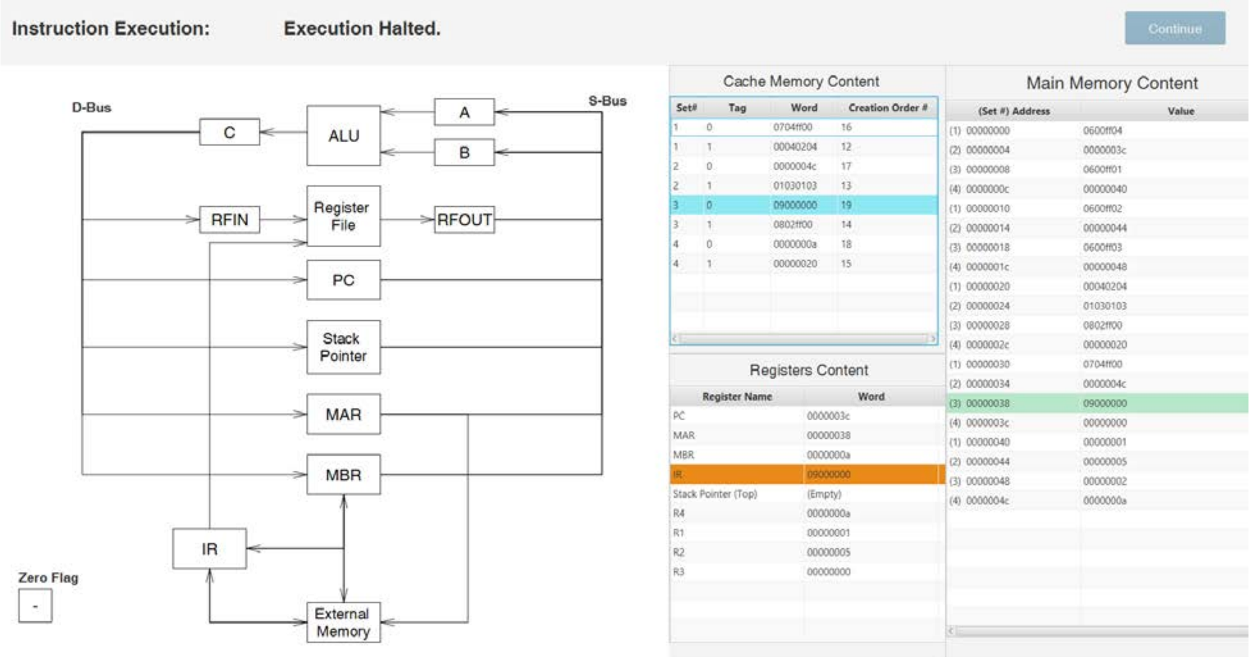
Main Memory Content

(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

Halt Operation

When the instruction is a HALT operation, the simulator will perform a write back operation for Cache Memory, i.e. update content on the corresponding main memory if modified flag is set true. Unlike other operations, the “Continue” button will be still remained “disabled”, i.e. cannot be clicked, even when the execution of halt operation is completed, as the simulation of the execution of the imported instruction set has already ended.

Example Screen Capture of HALT operation:



Instruction Execution: Execution Halted.

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0704ff00	16
1	1	00040204	12
2	0	0000004c	17
2	1	01030103	13
3	0	09000000	19
3	1	0802ff00	14
4	0	0000000a	18
4	1	00000020	15

Registers Content

Register Name	Word
PC	0000003c
MAR	00000038
MBR	0000000a
IR	09000000
Stack Pointer (Top)	(Empty)
R4	0000000a
R1	00000001
R2	00000005
R3	00000000

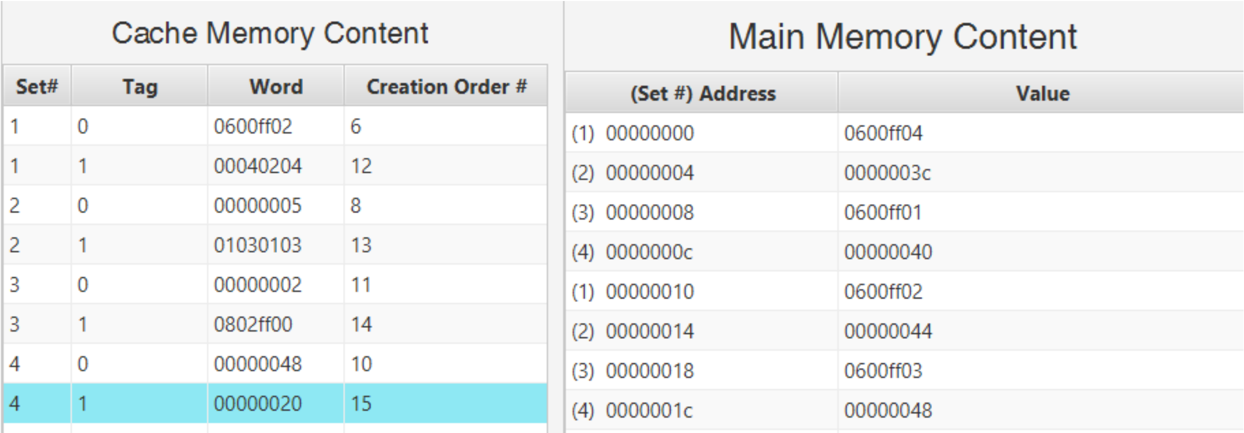
Main Memory Content

(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	0000000a

Cache Memory

Displaying Set Number in Main Memory Table

In real execution, the set number of memory address can be obtained from substring of binary form of the address, but in the Simulator for the purpose of demonstration, the set number is stated before the address in each row entry in the Main Memory table, thus users can quickly reference to the corresponding set in the Cache Memory Content table given the memory address to be accessed (which is explicitly stated in the description at the center of the top pane).



Cache Memory Content

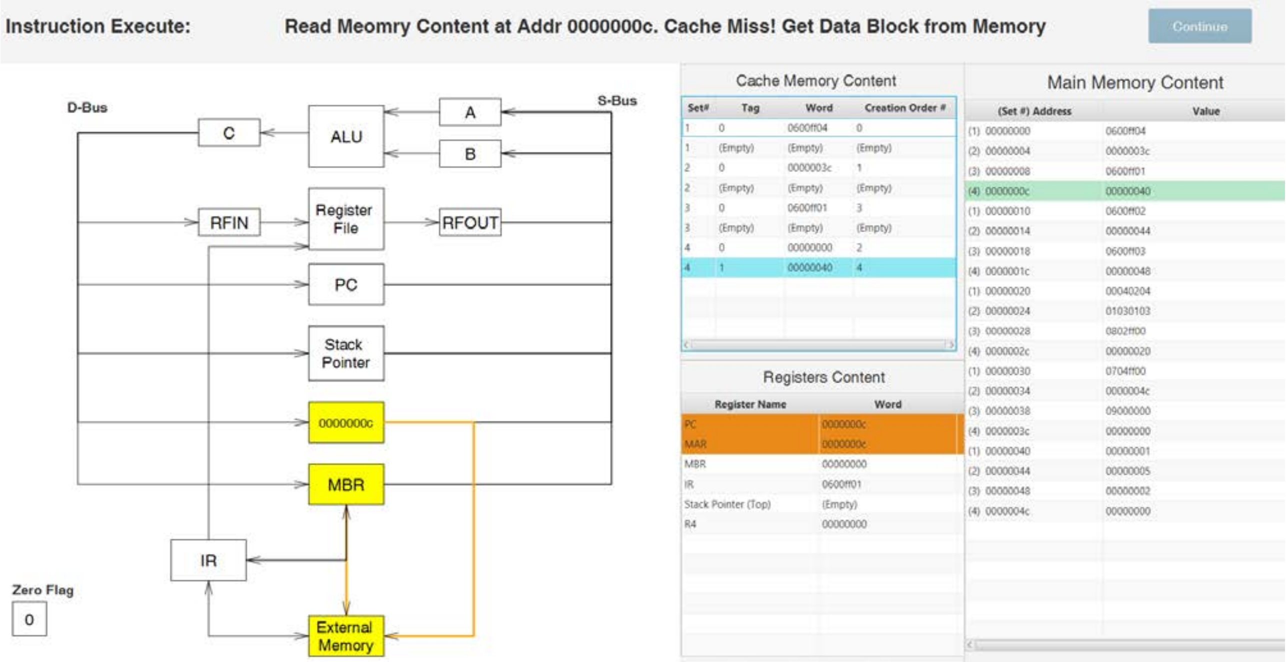
Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

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

Cache Miss (Without Cache Block Replacement)

This will be occurred when there is still vacancy cache entry line in the set. The description on the top center will be updated by stating that there is Cache Miss. At the same time the Cache Memory Content table will show the append of the new cache block. Appened cache block will be highlighted in light blue.



Instruction Execute: Read Meomry Content at Addr 0000000c. Cache Miss! Get Data Block from Memory

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff04	0
1	(Empty)	(Empty)	(Empty)
2	0	0000003c	1
2	(Empty)	(Empty)	(Empty)
3	0	0600ff01	3
3	(Empty)	(Empty)	(Empty)
4	0	00000000	2
4	1	00000040	4

Registers Content

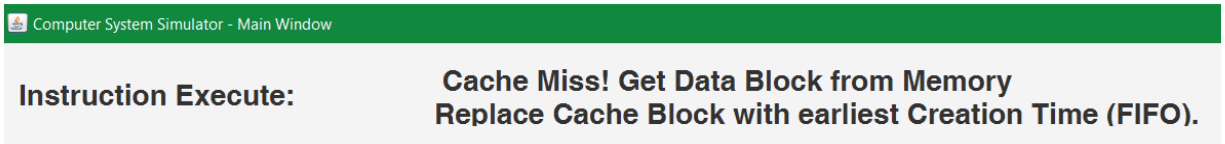
Register Name	Word
PC	0000000c
MAR	0000000c
MBR	00000000
IR	0600ff01
Stack Pointer (Top)	(Empty)
R4	00000000

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

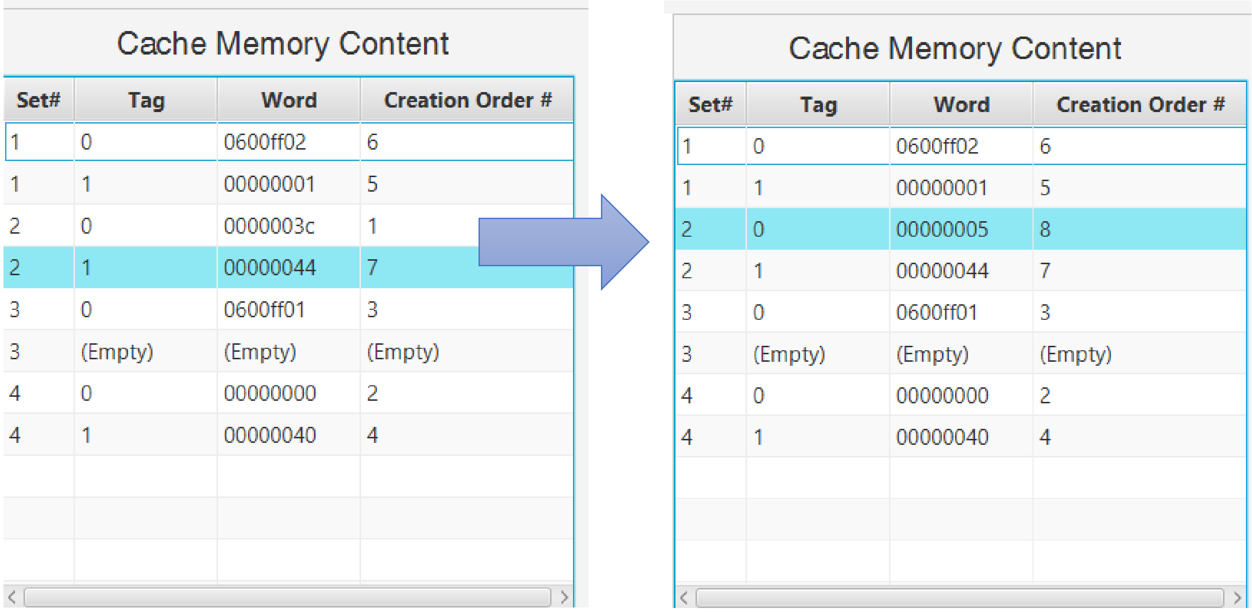
Cache Miss (With Cache Block Replacement)

When there is a Cache Miss that need cache block replacement, there will be one more line appended to the top center description compared with Cache Miss (Without Cache Block Replacement), the description notifies that which replacement algorithm is used, in the example of the below screen capture "FIFO" is used, and it is also stated the criteria for replacement, i.e. the Cache Block with the earliest creation time (i.e. "First-in") will be replaced.



Instruction Execute: Cache Miss! Get Data Block from Memory
Replace Cache Block with earliest Creation Time (FIFO).

The Cache Memory Content table will show the changes in the corresponding cache set, illustrated by the following screen capture:



Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00000001	5
2	0	0000003c	1
2	1	00000044	7
3	0	0600ff01	3
3	(Empty)	(Empty)	(Empty)
4	0	00000000	2
4	1	00000040	4

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00000001	5
2	0	00000005	8
2	1	00000044	7
3	0	0600ff01	3
3	(Empty)	(Empty)	(Empty)
4	0	00000000	2
4	1	00000040	4

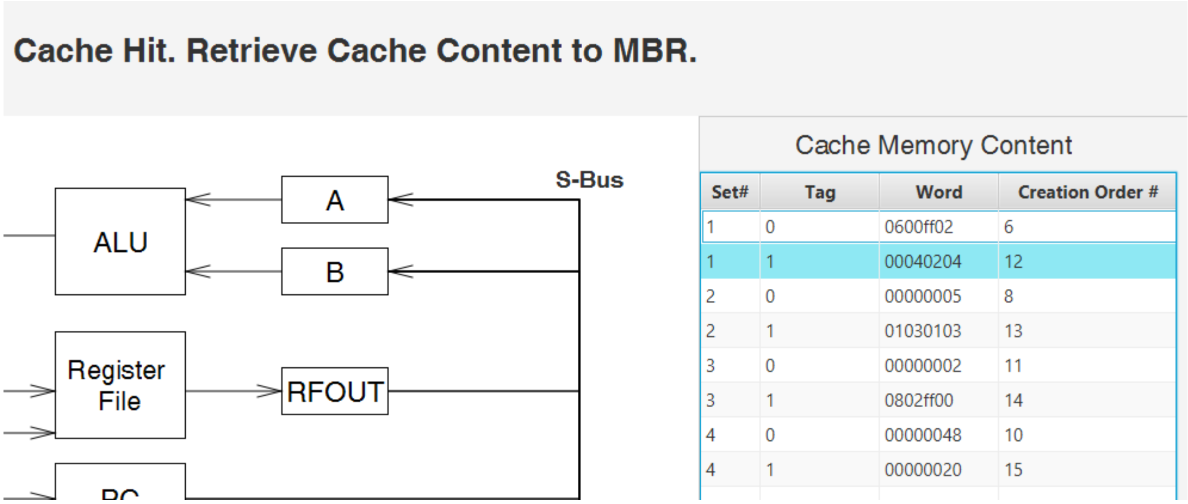
The left screen capture is the cache memory content before Cache Miss, the row highlighted in light-blue is the recently created cache block (recently access cache block for LRU).

The right screen capture shows the cache block replacement. The cache block should be added to Set 2, as FIFO replacement scheme is used in the screen capture's simulation, the row with earliest creation order in Set 2 is replaced.

The above screen capture is an example of 2-way associative cache with 4 sets. The tag of each set are all set as "0" and "1" for the purpose of showing a replacement of a cache block, as under this setting the location of the row will stay the same before and after the replacement.

Cache Hit

Cache Hit means that the data block to be retrieved is successfully found in one of the cache line block in the Cache Memory. In the case of a Cache Hit, the description at the top center will be updated to state that there is Cache Hit, and the corresponding row in the Cache Memory Content table will be highlighted in light-blue to indicate the cache block hit.

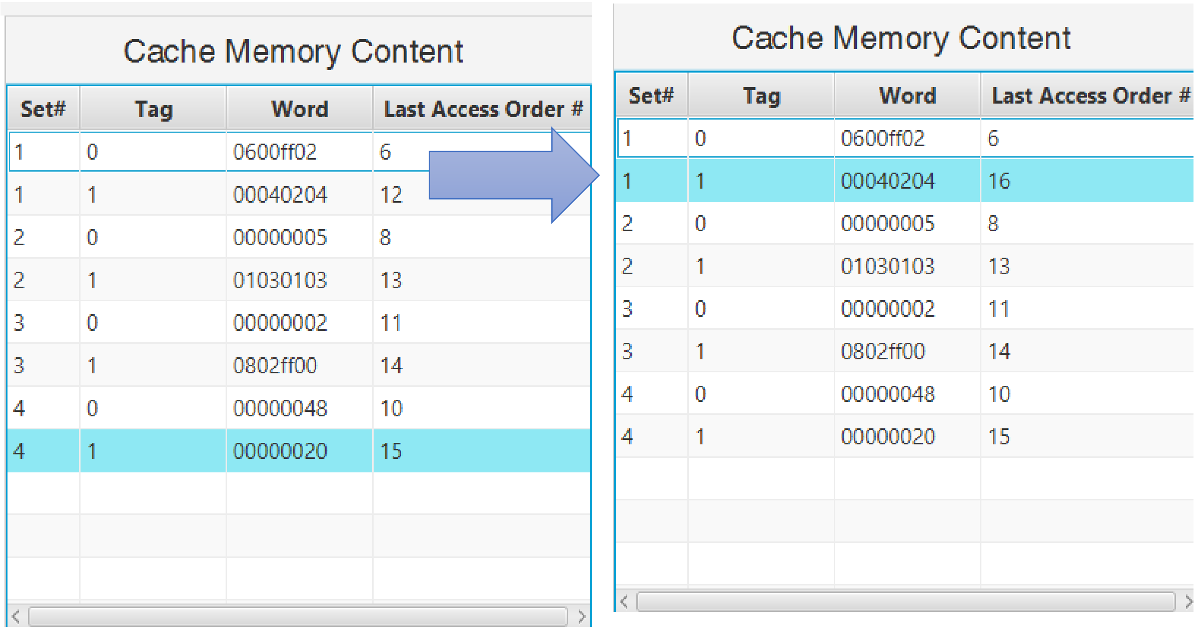


Cache Hit. Retrieve Cache Content to MBR.

Cache Memory Content

Set#	Tag	Word	Creation Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

When the Simulator is configured with replacement algorithm "LRU", the "Last Access Order #" column (i.e. the last column) of the cache block hit will be updated, illustrated as follows:



Cache Memory Content

Set#	Tag	Word	Last Access Order #
1	0	0600ff02	6
1	1	00040204	12
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

Cache Memory Content

Set#	Tag	Word	Last Access Order #
1	0	0600ff02	6
1	1	00040204	16
2	0	00000005	8
2	1	01030103	13
3	0	00000002	11
3	1	0802ff00	14
4	0	00000048	10
4	1	00000020	15

Future Works

Due to constraints in time and human resources, there are several possible future development for the simulator.

Pause for Every Step instead of Every Instruction

As mentioned in previous part, some instructions such as LOAD and STORE operation comprise of a relatively large amount of steps in one instruction execution; and for some steps there are several components changing together, thus there may have chances that the simulator already jumps to next step before users realize all the changes. However under the use of JavaFX, ordinary wait() and notify() does not work due to the difference of threading philosophy in JavaFX, due to the time constraint there was lack of research on the substitution of ordinary wait() and notify() in JavaFX, thus the simulator will enter idle state only when the execution of instruction is completed. Thus making the simulator idle after each step and continue next step when user trigger the continuation of the execution could be one of the major future works.

To compensate this limitation in the current implementation, users can modify the sleeping time between each step in the external configuration file (config.properties) if they find the pausing time between each step is too long or too short.

Backward Execution

Students may want to replay the previous step of the execution to have better understanding of the changes occurred, or to see whether he/she misses any of the changes, therefore backward execution would be an important function to let users to click a button to trigger the simulator to replay the previous step.

The implementation of this feature can be creation of a new class instance which has variables to store the main memory data list, cache data list, register file data list, value of zero flag and value of Program Counter at the beginning of the previous instruction execution. Making use of a stack data structure, the simulator program will push this object into a stack for each instruction execution, so when a user triggers a backward execution, the simulator program can pop out from the stack the previous data and re-assign the content of those variables in the main class and then continue the program execution.

Logging to External Text File

For command-line execution, any exceptions occurred during program execution can be logged as the stack trace for object that throws exceptions will be printed on the console output. However for GUI execution, there is no way to print the error stack trace for unexpected and uncaught exceptions, the GUI display will just freeze and user cannot realize immediately when the simulator program terminates its execution due to exceptions. Therefore there is need for the simulator to generate external log files for every execution of the simulator program (including execution of both Entry Window and Main Execution Window), so that any abnormal behavior of the simulator program can be reported by sending the log file to the party who hold the source code to perform debugging or source code amendments/enhancements.

Other Further Enhancement

There are other addressing mode, e.g. displacement mode, in instruction execution for some operations besides absolute addressing mode (i.e. 11111111), thus the simulator program could be further enhanced to cater those addressing mode that are normally covered in the Computer Organization course.

In addition, the current implementation can only cater single processor execution, also the execution steps are presumed to be executed one by one. However in reality, some instructions may be able to be processed in parallel in order to shorten the processing time, but the current implementation of the simulator does not take into account for the processing time of processor in the execution simulation.

Future works can be performed to enhance the functionality of the simulator to cater parallel processing as well as specify the execution order of instructions in the instruction set.

Conclusion

The structure and interconnection of operational units inside a computer are highly complex, thus understanding and teaching the data transformation and components interconnections just based on verbal descriptions and textbooks description is not easy. Thus there is apparent need for a simulator program to provide a more interactive and lively way to teach and learn such abstract concepts and ideas.

No single solution is prefect for all scenario. The way to solve a sub-problem may create constraints on other parts of the problem. This is what was frequently encountered throughout the development of the project, still with numerous research and refactoring, the final deliverables successfully meet all the objectives of the project. The Simulator covers processing units & memory hierarchy, is able to show interconnection of these components, with interactive graphical user interface, and with reasonable flexibility in configuration.

There are still rooms for enhancement for the final deliverable of this project, but it is definitely a great step forward from the existing teaching aids with greater coverage, flexibility and interactive elements with users. It is hoped that the Simulator will be used in the future and able to help students studying Computer Organisation in the future to understand the relevant concepts on data flow and transformation on different operational units in an instruction set execution in a more efficient and interesting way.

Appendix

Glossary

Abbreviation	Full Form/Description	Type
CPU	Central Processing Unit	Processor
CU	Control Unit	Component in CPU
ALU	Arithmetic and Logic Unit	Component in CPU
PC	Program Counter	Component in CU
IR	Instruction Register	Component in CU
MAR	Memory Address Register	Component in CU
MBR	Memory Buffer/Data Register	Component in CU
SP	Stack Pointer	Component in CU
RF	Register Files	Component in CU
RFIN	Register File Input Register	Component in CU
RFOUT	Register File Output Register	Component in CU
ADD	Addition	Instruction Operation
SUB	Subtraction	Instruction Operation
AND	Logic And	Instruction Operation
OR	Logic Or	Instruction Operation
NOT	Compliment	Instruction Operation
MOV/MOVE	Copy	Instruction Operation
BR	Branch (Always Branch)	Instruction Operation
BZ	Branch Zero (Branch if Zero Flag is set)	Instruction Operation
BNZ	Branch Not Zero (Branch if Zero Flag is not set)	Instruction Operation
LD/LOAD	Load data from memory to processor)	Instruction Operation
ST/STORE	Store data from processor to memory	Instruction Operation
HALT/HLT	Mark the completion of Instruction Set Execution	Instruction Operation
CALL	Function Call	Instruction Operation
RET	Function Return	Instruction Operation
PUSH	Push Register File to Register File List	Instruction Operation
POP	Pop Register File from Register File List	Instruction Operation
FIFO	First-In-First-Out	Cache Replacement Algorithm
LRU	Least Recently Use	Cache Replacement Algorithm

Simulator User Guide

The command-line version and GUI version of the Simulator can be downloaded from the Project Webpage.

How to run the Simulator-cmd.jar file:

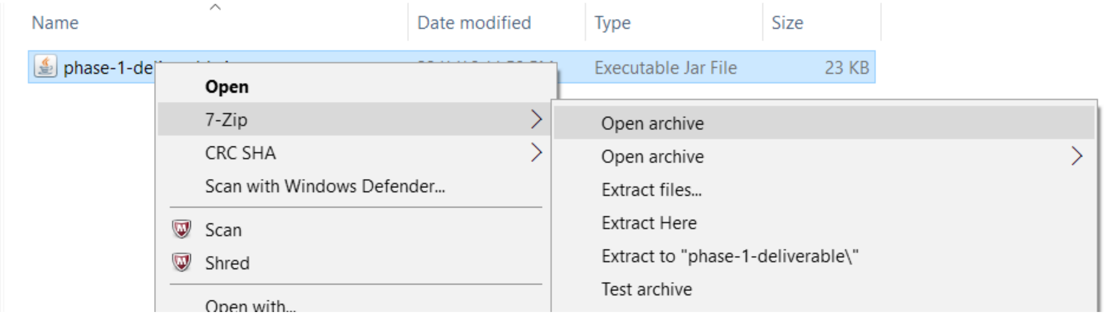
1. Open Command Prompt (for Windows) or Terminal (For Mac).
2. Type “java -jar (path of Simulator-cmd.jar file)”, e.g. if the file is saved in “C:\download”, then type “java -jar C:\download\ Simulator-cmd.jar”, and then press enter. Do not type the double quotation (“ ”).
3. If it failed to run, please check whether JRE (Java Runtime Environment) is installed in the machine. JRE can be downloaded here.

How to run the Simulator-gui.jar file:

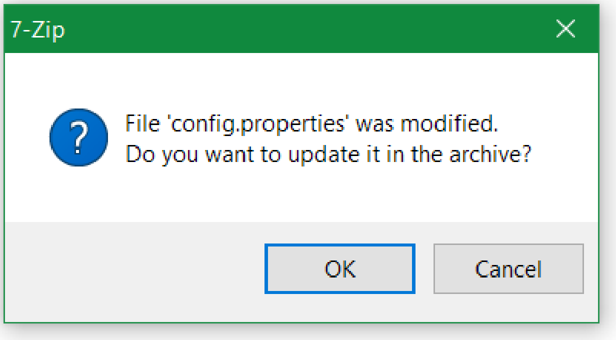
1. Just simply click the java executable file to run the GUI version Simulator.

How to change configuration:

1. Install 7-zip or other file archiver.
2. Go to the file directory of the .jar file.
3. Right Click the .jar file and choose “Open Archive”



4. Double click to open the .properties file that you want to alter. Recommend to choose to open with Notepad. Save after amendment.
5. Click “OK” if the following alert message prompt out:



File 'config.properties' was modified.
Do you want to update it in the archive?

6. Close the Archive Window before running the simulator.