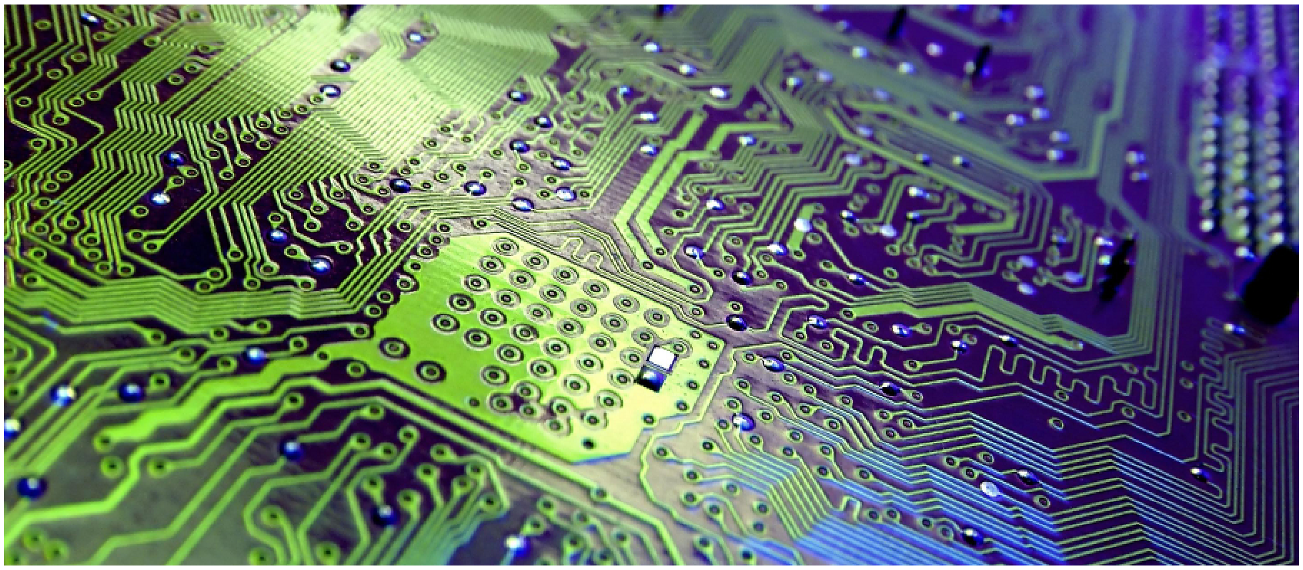
CSIS0801 Final Year Project

Phase 2 Deliverable

Interim Report

Topic: Computer System Simulator



Wong Jing Hin

UID: 3035051060

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.

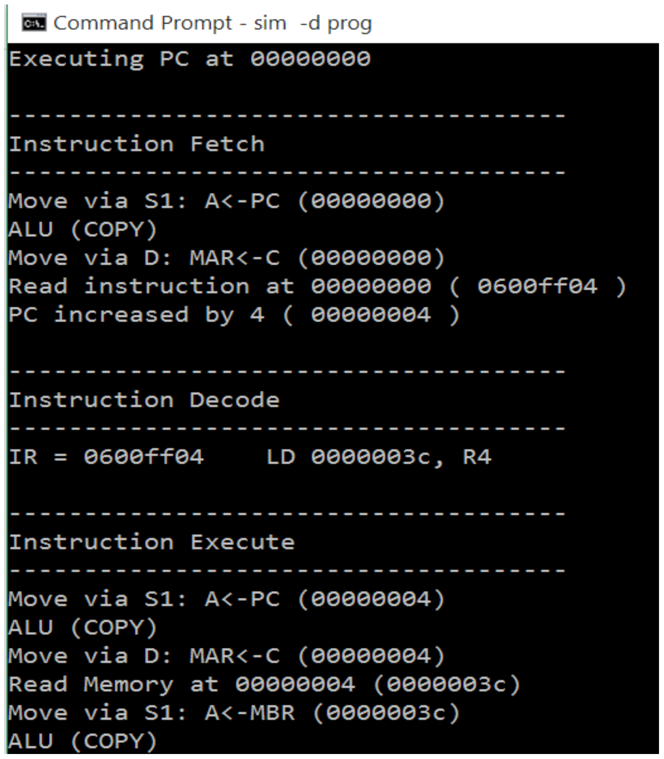
In an instruction execution, there are lots data movements among different components and transformations within components. Moreover, components are connected with each other differently, forming a complicated architecture of a computer system. Therefore, lots of undergraduate students find the flow 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:



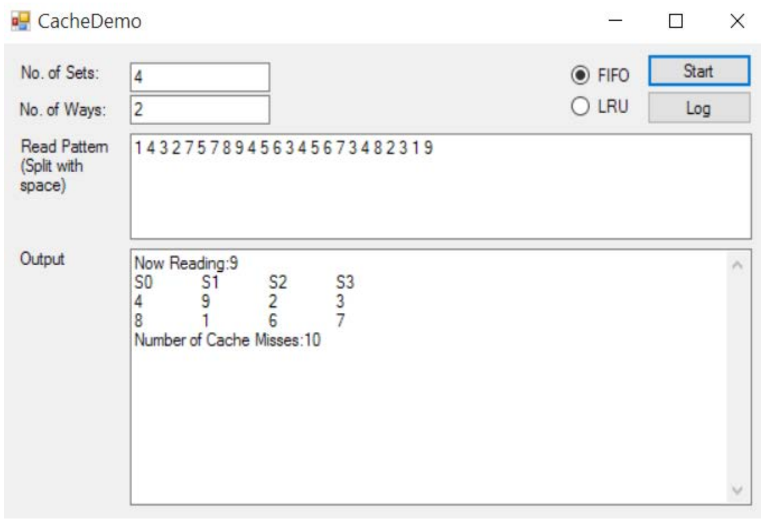
```
Executing PC at 00000000

------------------------------------
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 Objective

The primary objective of the project is 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 memory hierarchy (main memory & storage) will be covered in instruction set simulation of the simulator. 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)

One major drawback of the existing teaching aids programs are its command-line type display, thus the proposed simulator of this project will be developed with GUI. Components of the processor (e.g. registers, program counter and stack pointer), cache memory and memory hierarchy will be 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

Configuration will be independent of the core program source code. When a user runs the simulator, the simulator will automatically read 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.

Further Enhancement

There are other addressing mode in instruction execution besides absolute addressing mode e.g. displacement mode, thus the simulator will be further enhanced to cater those addressing mode that are normally covered in the Computer Organization course.

Project Methodology

Phased methodology will be 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 so 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 will be categorized into 4 phases. The first two phases will be the design and development of operational units, the order of design and development will be in the other way round of the accessing order of operation units in instruction set execution (e.g. if accessing order is: cache memory -> CPU, then development order will be: CPU -> cache memory), the latter phase will be extension of the deliverable from previous phase development. The development of the first 2 phases will be in command-line output, when the 2nd phase is completed, the core functionalities of the simulator will be considered as implemented with correct logic flow. Therefore for the 3rd stage, output format will be transformed into GUI form. The final phase (4th phase) will be further enhancements for the simulator as mentioned in the previous part.

Progress Report

Schedule

The completed or commenced tasks of the project until the 3rd week of January with the corresponding time period is shown in the following table:

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	1/11/15	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	10/1/15	6
8	Documentation for Phase 1	1/11/15	*(on-going)*	3
9	Phase 2 (Cache Memory) Analysis & Design	11/1/16	29/1/15 *(expected)*	7
10	**Milestone: First Presentation**	**12/1/16**	**n/a**	

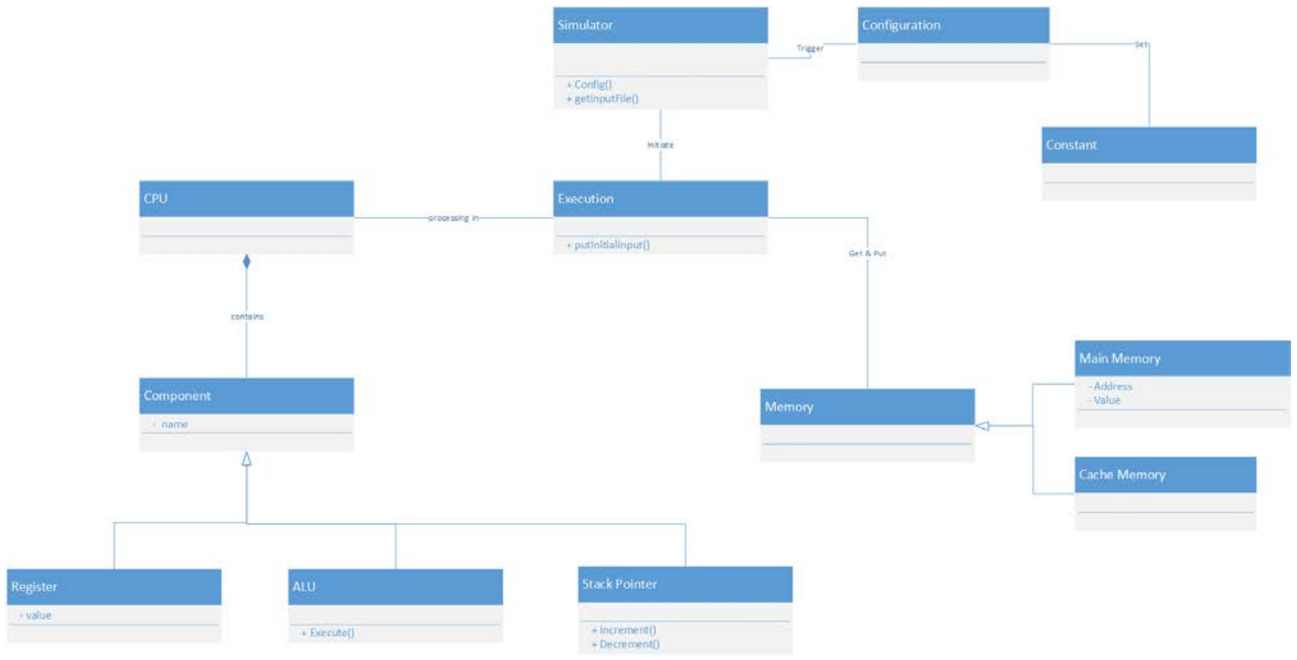
The deliverables for Phase 1 can be downloaded from the Project Website.

The table below shows the expected schedule for the 2nd semester work:

ID	Task Name	Expected Start Date	Expected Duration	Predecessors
11	**Milestone: Interim Report Submission**	**24/1/16**	**n/a**	
12	Phase 2 Coding & Testing	30/1/16	3 weeks	9
13	Phase 2 Documentation	30/1/16	*Till 15/4/16*	7
14	Phase 3 (GUI) Implementation	15/2/16	4 weeks	12
15	Phase 4 (Enhancement) Analysis & Design	14/3/16	1 weeks	14
s16	Phase 4 Coding & Testing	21/3/16	3 weeks	15
17	Integrated Testing	28/3/16	3 weeks	14
18	Documentation for Phase 3&4	15/2/16	*Till 15/4/16*	12
19	**Milestone: Final Report Submission**	**17/4/16**	**n/a**	

1st Semester Accomplishment

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 involve processing and storage, thus there are 2 class, CPU & Memory, expanded from the Execution class.

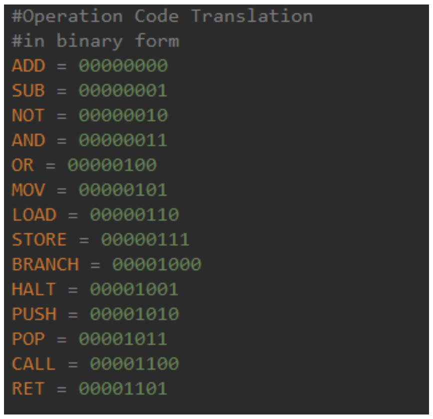
For Memory class, there are 2 types of memory involved, thus Main Memory & Cache Memory class object are created, which extends the Memory Object.

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.

Algorithm Design

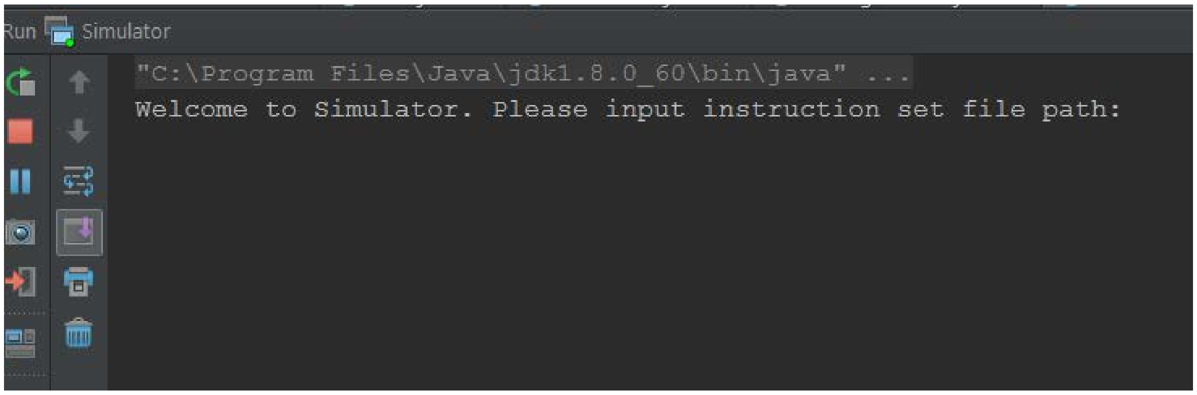
After start running the Simulator:

The main method in the Simulator will run, new configuration class will be created which trigger the configuration setting for the simulator and execution. The right is the screen capture of the property file for operation code definition. It is an external text file, which can be modified by any user.



```
#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
```

Then the simulator will ask user to input the file path of the instruction set to be demonstrated. Below is the sample command-line console output of this process of the simulator:

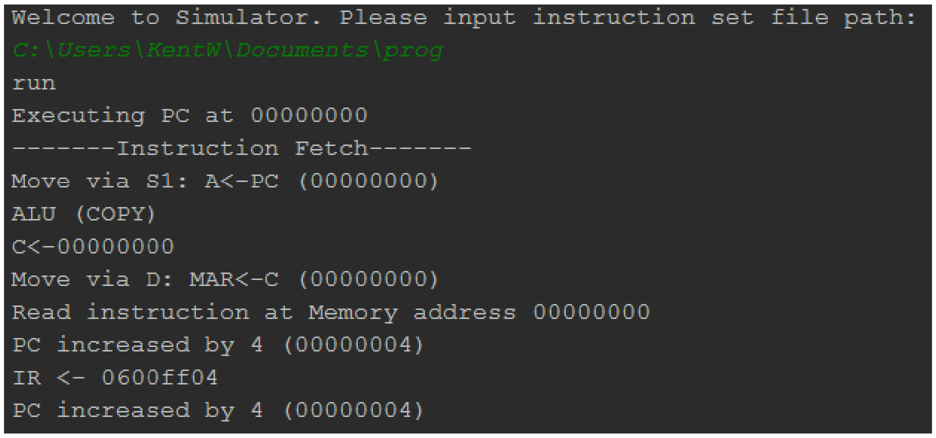


```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...
Welcome to Simulator. Please input instruction set file path:
```

Execution

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.

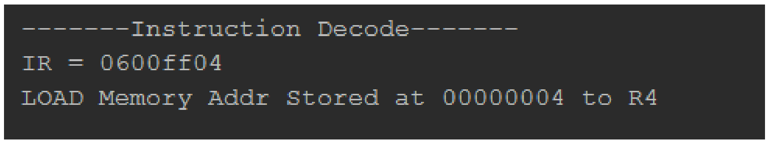
Next the execution of instruction set starts. The execution will be simulated by performing unconditional looping, until a HALT instruction is detected and exit the loop by "break" command. For each loop, the algorithm quite resembles a real instruction execution. Firstly the "Program Counter" object 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.



```
Welcome to Simulator. Please input instruction set file path:
C:\Users\KentW\Documents\prog
run
Executing PC at 00000000
-------Instruction Fetch-------
Move via S1: A<-PC (00000000)
ALU (COPY)
C<-00000000
Move via D: MAR<-C (00000000)
Read instruction at Memory address 00000000
PC increased by 4 (00000004)
IR <- 0600ff04
PC increased by 4 (00000004)
```

Figure 1: Command-line Output of "Instruction Fetch"

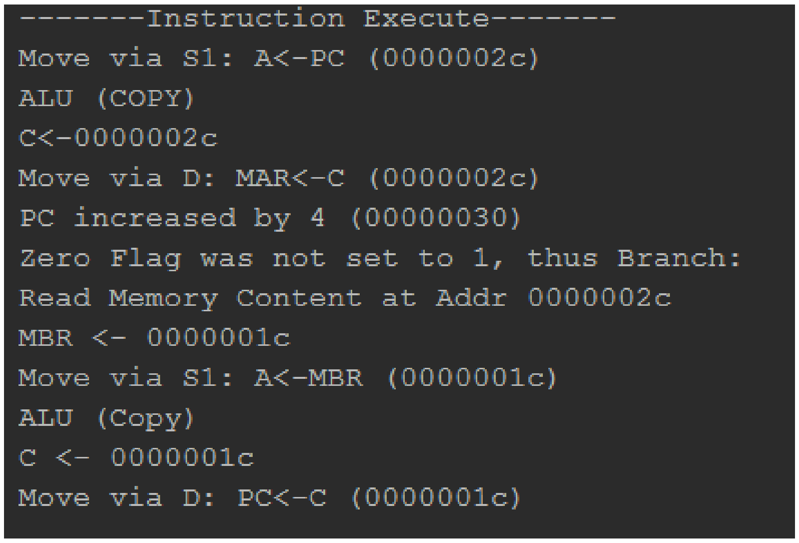
Next will be instruction decoding. The content stored in "Instruction Register" object will be extracted and analyzed. The string will be divided into 4 sub-string, the 1st sub-string indicate the operation code, the program will then map the operation code with the "definition table" set from configuration,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



```
-------Instruction Decode-------
IR = 0600ff04
LOAD Memory Addr Stored at 00000004 to R4
```

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). Below is a sample output in command-line form for a branch operation (Branch Not Zero type):

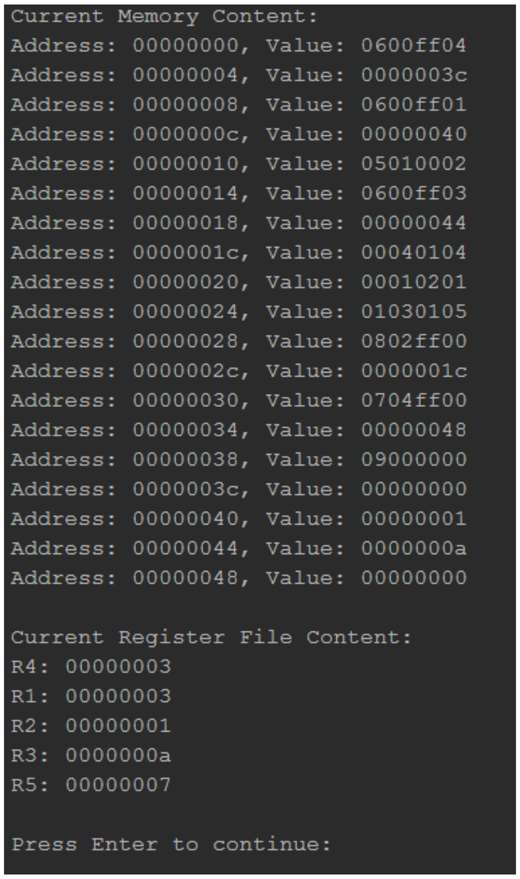


```
-------Instruction Execute-------
Move via S1: A<-PC (0000002c)
ALU (COPY)
C<-0000002c
Move via D: MAR<-C (0000002c)
PC increased by 4 (00000030)
Zero Flag was not set to 1, thus Branch:
Read Memory Content at Addr 0000002c
MBR <- 0000001c
Move via S1: A<-MBR (0000001c)
ALU (Copy)
C <- 0000001c
Move via D: PC<-C (0000001c)
```

Every output statement will be transformed into graphical illustration in Phase 3 (GUI Implementation).

For GUI form output, content of memory and register files will be shown all the time throughout the whole execution simulation. At the current stage the output is still in form of command-line, thus currently at the end of each instruction execution, the content of memory and register files will be displayed to resemble such feature.

At the last line of the command-line output on the right figure, it asks user to "Press Enter to Continue". After pressing enter, the simulator will continue to execute the next instruction, if there is no more next instruction, i.e. the current instruction is HALT, the simulation will come to an end, and showing the final content of the Memory. For GUI display, there will be a button to serve this purpose.



```
Current Memory Content:
Address: 00000000, Value: 0600ff04
Address: 00000004, Value: 0000003c
Address: 00000008, Value: 0600ff01
Address: 0000000c, Value: 00000040
Address: 00000010, Value: 05010002
Address: 00000014, Value: 0600ff03
Address: 00000018, Value: 00000044
Address: 0000001c, Value: 00040104
Address: 00000020, Value: 00010201
Address: 00000024, Value: 01030105
Address: 00000028, Value: 0802ff00
Address: 0000002c, Value: 0000001c
Address: 00000030, Value: 0704ff00
Address: 00000034, Value: 00000048
Address: 00000038, Value: 09000000
Address: 0000003c, Value: 00000000
Address: 00000040, Value: 00000001
Address: 00000044, Value: 0000000a
Address: 00000048, Value: 00000000

Current Register File Content:
R4: 00000003
R1: 00000003
R2: 00000001
R3: 0000000a
R5: 00000007

Press Enter to continue:
```

Pseudocode Showcase

1. Execution of Instruction Set

```
repeat
begin
        Address <- PC.value
        Instruction <- Memory[Address]
        PC.value <- PC.value + 4
        Analyze_Instruction(Instruction)
end
until (Instruction.OperationCode = HALT)
```

2. Analyze_Insturction

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

3. Add/Sub/And/Or

```
SourceRegister1Name <- SepOp.substring(2nd Part)
Source1Value <- GetValueOfRegisterOfName(SourceRegister1Name)
SourceRegister2Name <- SepOp.substring(3rd Part
Source2Value <- GetValueOfRegisterOfName(SourceRegister2Name)
DestinationRegisterName <- SepOp.substring(4th Part)
DestValue <- ALU.add/sub/and/or (source1Value, source2Value)
UpdateRegisterFile(DestinationRegisterName, DestValue)
```

(For "Not" & "Move" Operation, same pseudocode as above except there will be no lines involving Source Register 2)

4. ALU Operation (Add/Sub/And/Or/Not/Move)

```
Operation (Value1, Value2)
begin
        ReturnVal <- Value1 + Value2/ Value1 - Value2/ Value1 && Value2/ etc.
        If (returnVal == 0)
        then ZeroFlag <- 1
        else ZeroFlag <- 0
        return ReturnVal
end
```

5. Load

```
LoadRegisterName <- SepOp.substring(4th Part)
NextMemoryAddress <- PC.value
NextMemoryContent <- Memory[NextMemoryAddress]
PC.value <- PC.value + 4
LoadValue <- Memory[NextMemoryContent]
UpdateRegisterFile(LoadRegisterName, LoadValue)
```

6. Store

```
StoreRegisterName <- SepOp.substring(2nd Part)
NextMemoryAddress <- PC.value
NextMemoryContent <- Memory[NextMemoryAddress]
StoreValue <- GetValueOfRegisterOfName(StoreRegisterName)
PC.value <- PC.value + 4
Memory[NextMemoryContent] <- StoreValue
```

7. Branch (BR/BZ/BNZ)

```
NextMemoryAddress <- PC.value
NextMemoryContent <- Memory[NextMemoryAddress]
PC.value <- Memory[NextMemoryContent]
[Branch Zero (BZ)]
if (ALU.ZeroFlag != 0)
then PC.value <- PC.value + 4
[/BZ]
[Branch Not Zero (BNZ)]
if (ALU.ZeroFlag == 0)
then PC.value <- PC.value + 4
[/BNZ]
```

8. Call (Function Call)

```
StackPointer.Increment (PC.value + 4)
NextMemoryAddress <- PC.value
NextMemoryContent <- Memory[NextMemoryAddress]
PC.value <- Memory[NextMemoryContent]
```

9. Return (of Function Call)

```
NextAddress <- StackPointer.Decrement()
PC.value <- NextAddress
```

10. Push (of Function Call)

```
PushRegisterName <- SepOp.substring(2nd Part)
AddNewRegisterFile(PushRegisterName)
```

11. Pop (of Function Call)

```
PopRegisterName <- SepOp.substring(4th Part)
RemoveRegisterFile(PushRegisterName)
```

12. AddNewRegisterFile

```
NewRegister = new Register()
NewRegsiter.name <- PushRegisterName
RegisterFileList.add(NewRegister)
```

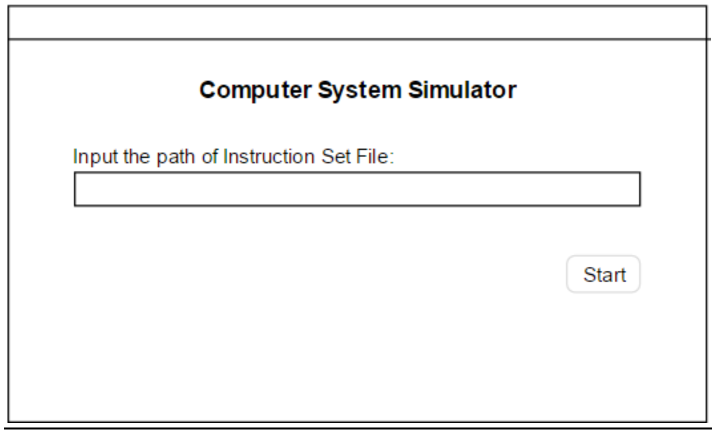
13. RemoveRegisterFile

```
RemoveIndex <- GetIndexOfRegisterFileInRegisterFileList(PopRegisterName)
RegisterFileList.remove(RemoveIndex)
```

UI Design

The user-interface design for Phase 3: GUI Implementation has been done, which are displayed below:

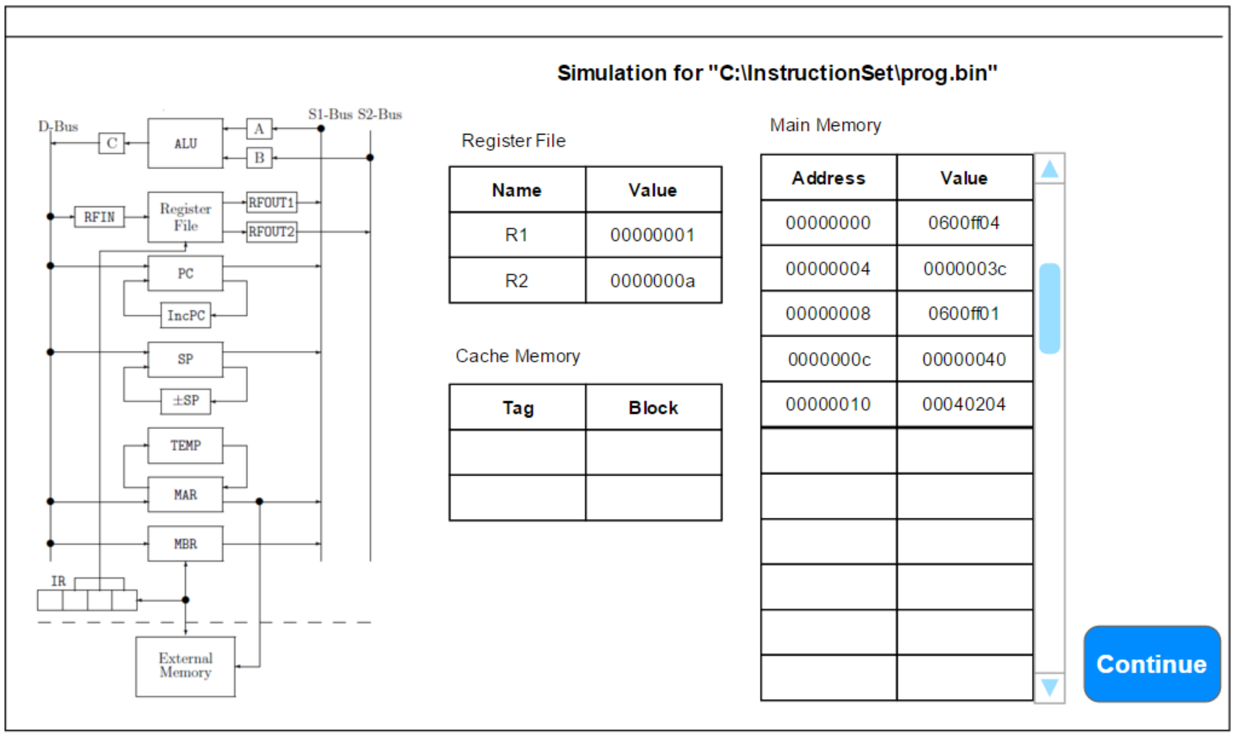
1. *Initial (Index) Window*



Computer System Simulator

Input the path of Instruction Set File:

2. *Base*



Simulation for "C:\InstructionSet\prog.bin"

Register File

Name	Value
R1	00000001
R2	0000000a

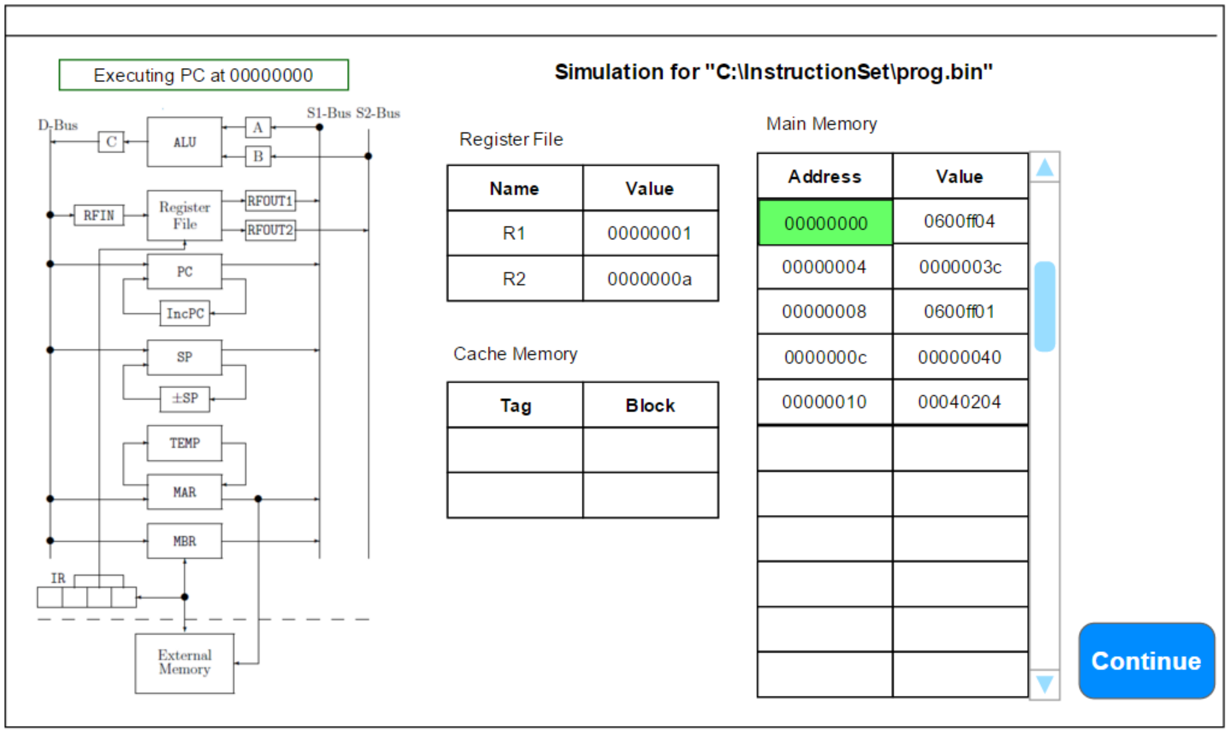
Cache Memory

Tag	Block

Main Memory

Address	Value
00000000	0600ff04
00000004	0000003c
00000008	0600ff01
0000000c	00000040
00000010	00040204

3. *Running an Execution*



Simulation for "C:\InstructionSet\prog.bin"

Register File

Name	Value
R1	00000001
R2	0000000a

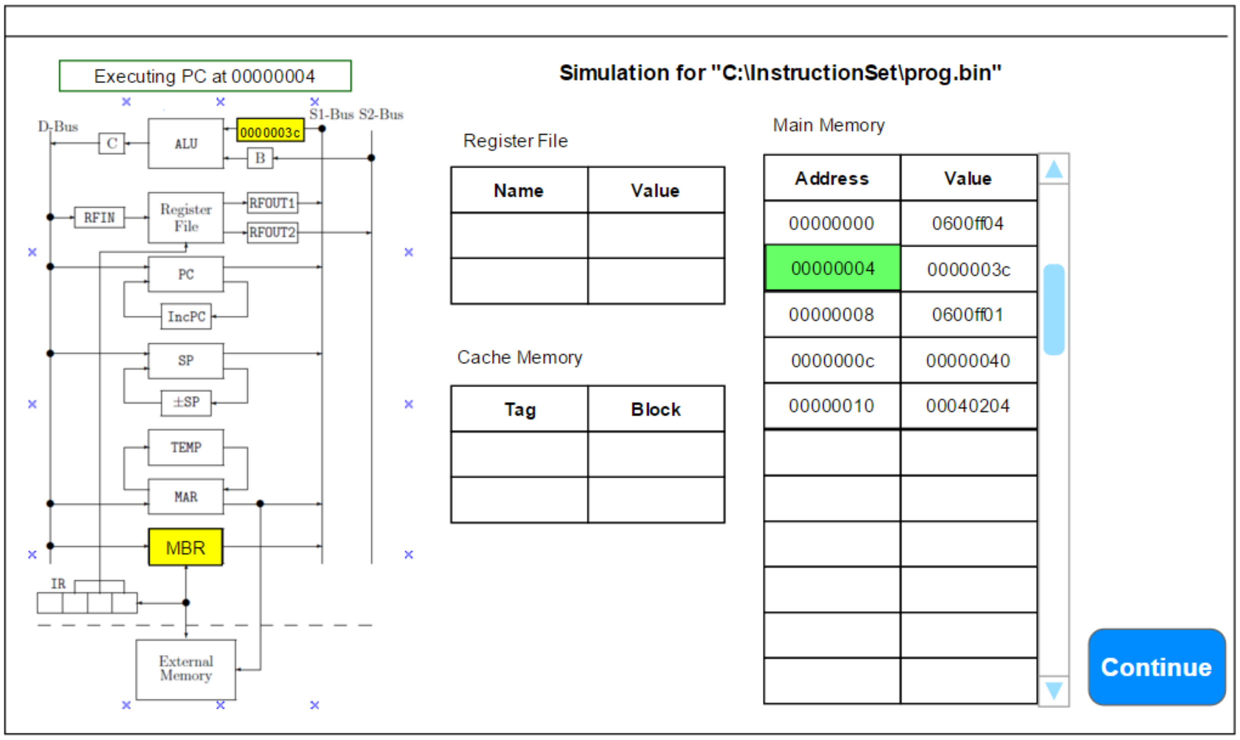
Cache Memory

Tag	Block

Main Memory

Address	Value
00000000	0600ff04
00000004	0000003c
00000008	0600ff01
0000000c	00000040
00000010	00040204

4. *Indicating Data Transformation & Transfer*
   (e.g. Data Movement from MBR to Register A)



Simulation for "C:\InstructionSet\prog.bin"

Register File

Name	Value

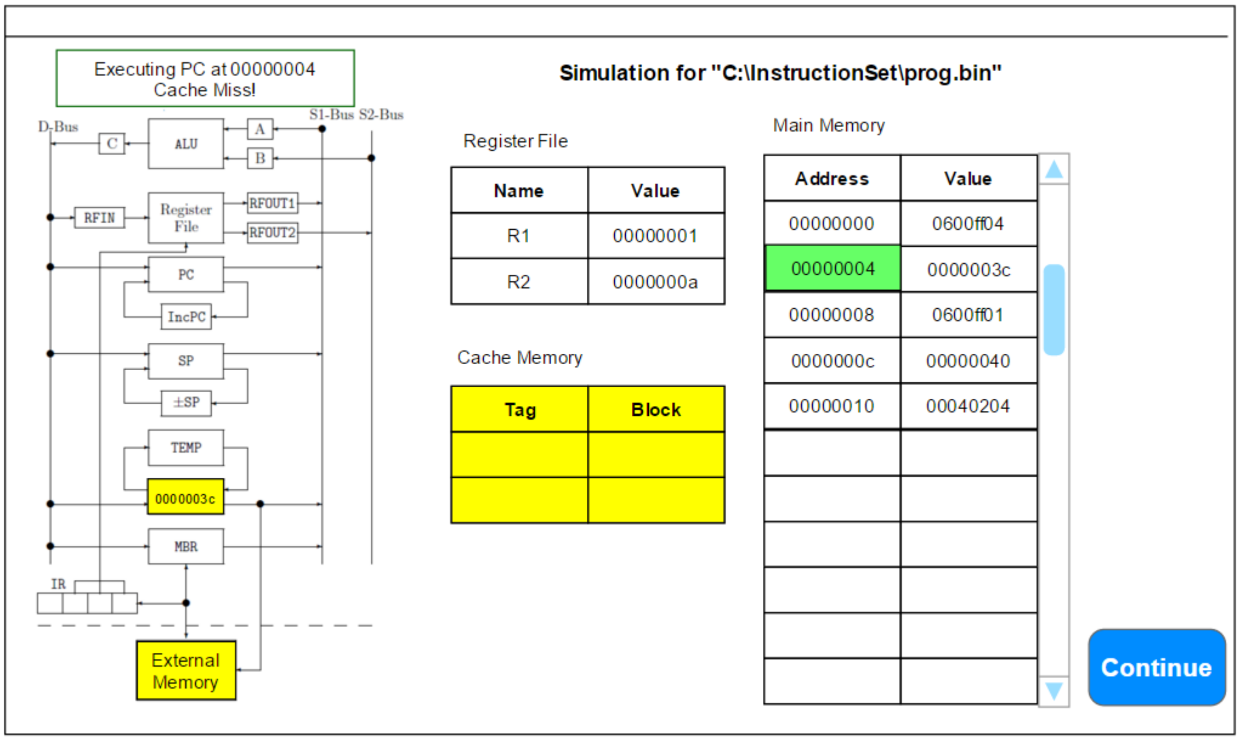
Cache Memory

Tag	Block

Main Memory

Address	Value
00000000	0600ff04
00000004	0000003c
00000008	0600ff01
0000000c	00000040
00000010	00040204

5. *Indicating Cache Miss*
(e.g. “0000003c” stored in MAR, processor has to read content in memory address 0000003c from External Memory)



Executing PC at 00000004
Cache Miss!

Simulation for "C:\InstructionSet\prog.bin"

Register File

Name	Value
R1	00000001
R2	0000000a

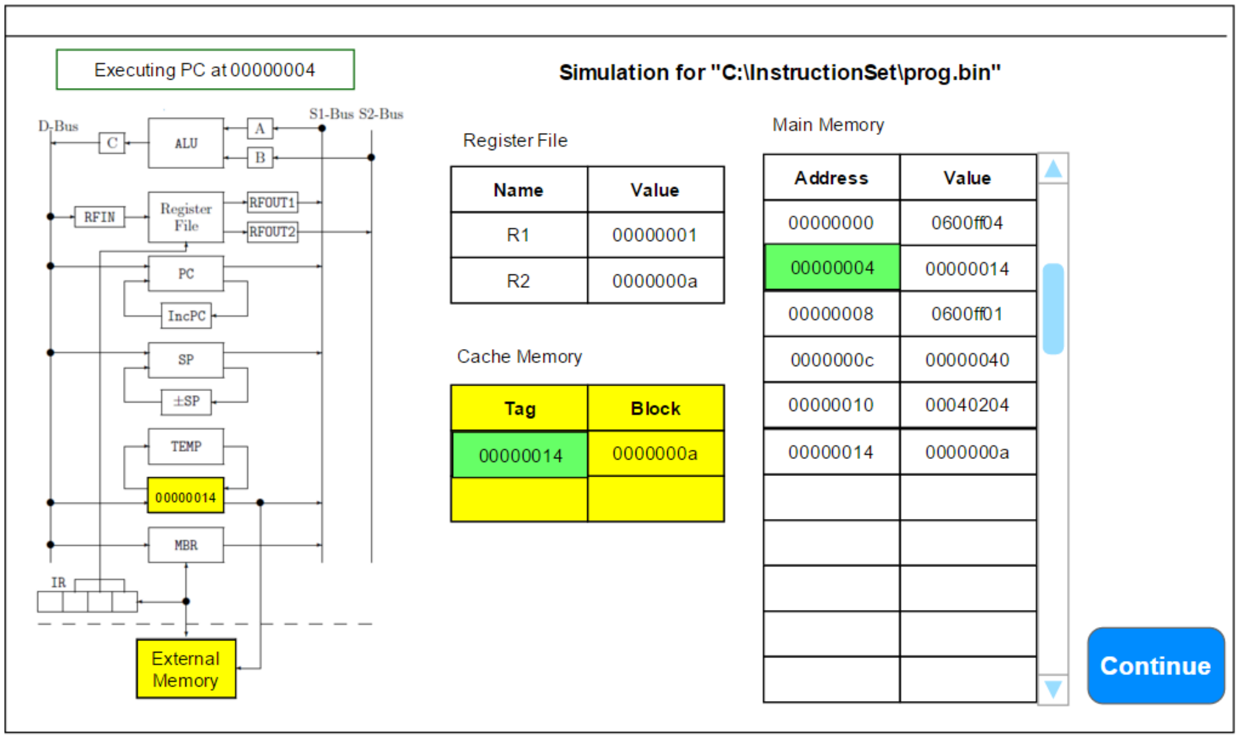
Cache Memory

Tag	Block

Main Memory

Address	Value
00000000	0600ff04
00000004	0000003c
00000008	0600ff01
0000000c	00000040
00000010	00040204

6. *Indicating Cache Hit*
(e.g. “00000014” stored in MAR, processor has to read content in memory address 00000014 from External Memory)



Executing PC at 00000004

Simulation for "C:\InstructionSet\prog.bin"

Register File

Name	Value
R1	00000001
R2	0000000a

Cache Memory

Tag	Block
00000014	0000000a

Main Memory

Address	Value
00000000	0600ff04
00000004	00000014
00000008	0600ff01
0000000c	00000040
00000010	00040204
00000014	0000000a

End of Interim Report