User Manual of the Simulator

Overview

This simulator is designed to help students better understand the principle operations in the CPU when learning and programming with the instruction set as well as follow closely the different cache replacement algorithms, by providing them with a visualization of the operations.

To use this simulator, users have to first define or import a set of instruction definitions. Then they can utilize the operation codes defined to construct the hexadecimal programme input in the simulation interface.

To start the simulator program

Enter the follow command in the command line tool at the directory containing the Java simulator program:

```
java –jar Simulator.jar
```

Note that Java 1.8 or above is required for this simulator program.

To define instructions

The following lists the functionalities of different interface components as shown in figure 1 for defining instructions.

#	Function
1	Remove the instruction
2	Mnemonic symbol that has to be unique, maximum 6 characters
3	Operation code that has to be unique, 2 hexadecimal digits
4	Show/Hide the definition configuration panel on the right
5	Add an instruction, maximum 16 instructions
6	Import instruction definitions from a definition txt file

#	Function
7	Export currently saved instructions as a definition txt file
8	Addressing mode under which this action will run, "*" for all modes
9	Add an action in parallel within the same execution timeframe
10	Choose from 22 elementary actions
11	Supplementary configurations for this action, including bus source and destination for system bus-related actions, and set-flag option for ALU-related actions
12	Add next timeframe
13	Remove the last timeframe, including all the parallel actions within that timeframe. Individual removal of timeframes that are not the last is not supported.
14	Save this instruction definition

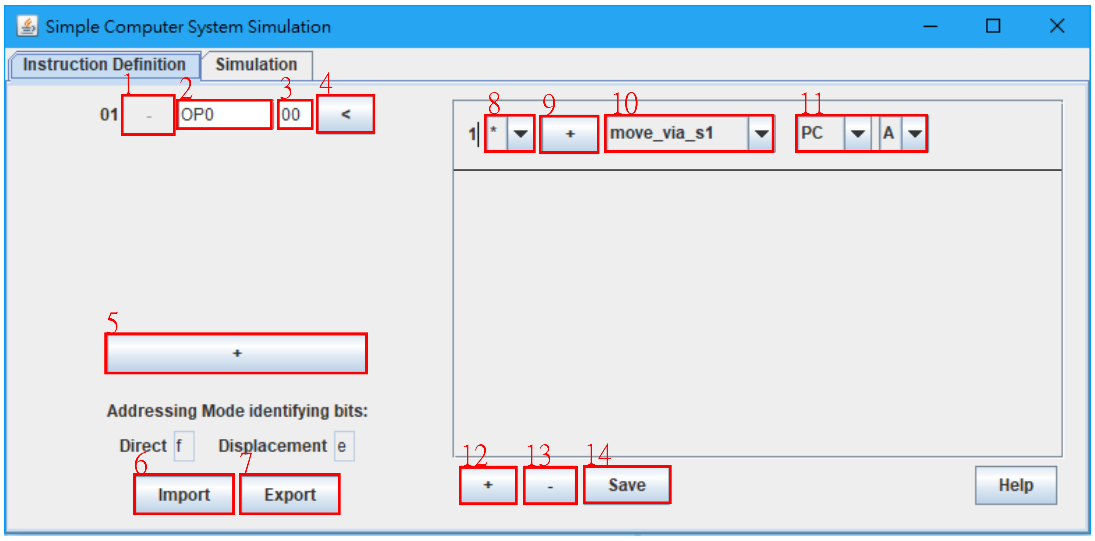

Figure 1

The key to effectively construct instructions is to know the individual end-effect of each and every of the 22 elementary hardware-level actions, that are used in different combinations and sequences to achieve the overall function of the instruction defined. The functions of the actions are listed as follows.

Action	Function
move_via_s1	1. update value of the data point from the data source 2. move the data point from the data source to the destination along the S1-Bus 3. update value of the destination with the value from the data point object

Action	Function
move_via_s2	same as move_via_s1, except S2-Bus is used instead of S1-Bus
move_via_d	same as move_via_s1, except D-Bus is used instead of S1-Bus
inc_pc	1. update value of the data point from PC 2. move the data point from PC to IncPC 3. increment value of data point by 4 4. move the data point from IncPC to PC 5. update the value of PC from the data point
read_rf_port1	1. update value of the data point from the register file specified in source operand 1 2. move the data point from Register File to RFOUT1 3. update value of RFOUT1 from the data point
read_rf_port2	same as read_rf_port1, except register file specified in source operand 2 is used instead * note that only the 5th bit in the instruction word is significant in specifying the register file
write_rf	1. update value of the data point from RFIN 2. move the data point from RFIN to Register File 3. update value of the specified register file from the data point
alu_add	1. update values of the data points from temporary registers A and B respectively 2. move the data points from A and B to ALU simultaneously 3. perform addition on the values of the data points 4. update value of the data point from A with the result after the ALU operation, discard the other data point 5. move the data point from ALU to temporary register C 6. update value of C from the data point
alu_sub	same as alu_add, except the ALU operation is substraction, i.e. "A" - "B"
alu_and	same as alu_add, except the ALU operation is bit-wise AND
alu_or	same as alu_add, except the ALU operation is bit-wise OR

Action	Function
alu_not	1. update values of the data point from temporary register A 2. move the data point from A to ALU 3. perform bit-wise NOT on the value of the data point 4. update value of the data point with the result after the ALU operation 5. move the data point from ALU to temporary register C 6. update value of C from the data point
alu_copy	1. update values of the data point from temporary register A 2. move the data point from A to ALU 3. move the data point from ALU to temporary register C 4. update value of C from the data point
read_instruction	1. update value of the data point from MAR 2. move the data point from MAR to External Memory 3. read memory content with the address data in the data point 4. update value of the data point with the data from the memory read 5. move the data point from External Memory to IR 6. update value of IR from the data point 7. move the data point from IR to Register File to indicate preparation of the involved register files
read_memory	1. update value of the data point from MAR 2. move the data point from MAR to External Memory 3. read memory content with the address data in the data point 4. update value of the data point with the data from the memory read 5. move the data point from External Memory to MBR 6. update value of MBR from the data point
write_memory	1. update values of the data points from MAR and MBR respectively 2. move the data points from MAR and MBR to External Memory 3. update value of the simulated memory at the corresponding address, i.e. data in MAR, with the specified data, i.e. data in MBR

Action	Function
branch	1. check for branching condition, if branch, do the following, else continue to the next operations 2. update value of the data point from MBR 3. move the data point from MBR to temporary register A 4. update value of A from the data point 5. perform alu_copy 6. perform move_via_d from C to PC 7. skip all remaining operations in the current instruction and continue to the next instruction
dec_sp	1. update value of the data point from SP 2. move the data point from SP to +-SP 3. decrement value of data point by four 4. move the data point from +-SP to SP 5. update the value of SP from the data point
inc_sp	same as dec_sp, except increment SP instead of decrement
mar_to_temp	1. update value of the data point from MAR 2. move the data point from MAR to TEMP 3. update the value of TEMP from the data point
temp_to_mar	1. update value of the data point from TEMP 2. move the data point from TEMP to MAR 3. update the value of MAR from the data point
halt	halt the simulation * note that halt is designed specifically for the simulator to break out of the execution loop, and is not seen in real-life CPU

Animations of the respective actions can be found at http://i.cs.hku.hk/fyp/2016/fyp16026/

In addition to knowing what each of the elementary actions will do, the other important point to pay attention to is avoiding resource conflict when configuring actions in parallel.

The following shows the respective resources occupied when running the actions. When two actions occupying the same resource are configured in parallel, error occurs and the definition will be denied.

Action	Resources in use
move_via_s1	Bus Source, Bus Destination, S1-Bus
move_via_s2	Bus Source, Bus Destination, S2-Bus
move_via_d	Bus Source, Bus Destination, D-Bus
inc_pc	PC, IncPC
read_rf_port1	RF1
read_rf_port2	RF2
write_rf	RFIN, RF
alu_add	A, B, ALU, C
alu_sub	A, B, ALU, C
alu_and	A, B, ALU, C
alu_or	A, B, ALU, C
alu_not	A, ALU, C
alu_copy	A, ALU, C
read_instruction	MAR, External Memory, IR, RF
read_memory	MAR, External Memory, MBR
write_memory	MAR, External Memory, MBR
branch	MBR, S1-Bus, A, ALU, C, PC
dec_sp	SP, +-SP
inc_sp	SP, +-SP
mar_to_temp	MAR, TEMP
temp_to_mar	MAR, TEMP
halt	-

To simulate programme execution

The following lists the functionalities of different interface components as shown in figure 2 for beginning the graphical simulation of the programme input.

#	Function
1	Enable the cache memory simulation interface
2	Display of data in the registers, default in hexadecimal

#	Function
3	Switch the display bases between decimal and hexadecimal
4	Text area for inputting hexadecimal machine codes for the simulation, execution will start at "00000" and will terminate on "halt" or undefined memory
5	Translate the machine code into assembly code
6	Change the speed of the simulation animation
7	Begin/Pause the simulation
8	Stop and reset the simulation
9	Descriptive text field showing current CPU action being simulated
10	Descriptive text field showing current line of code being simulated

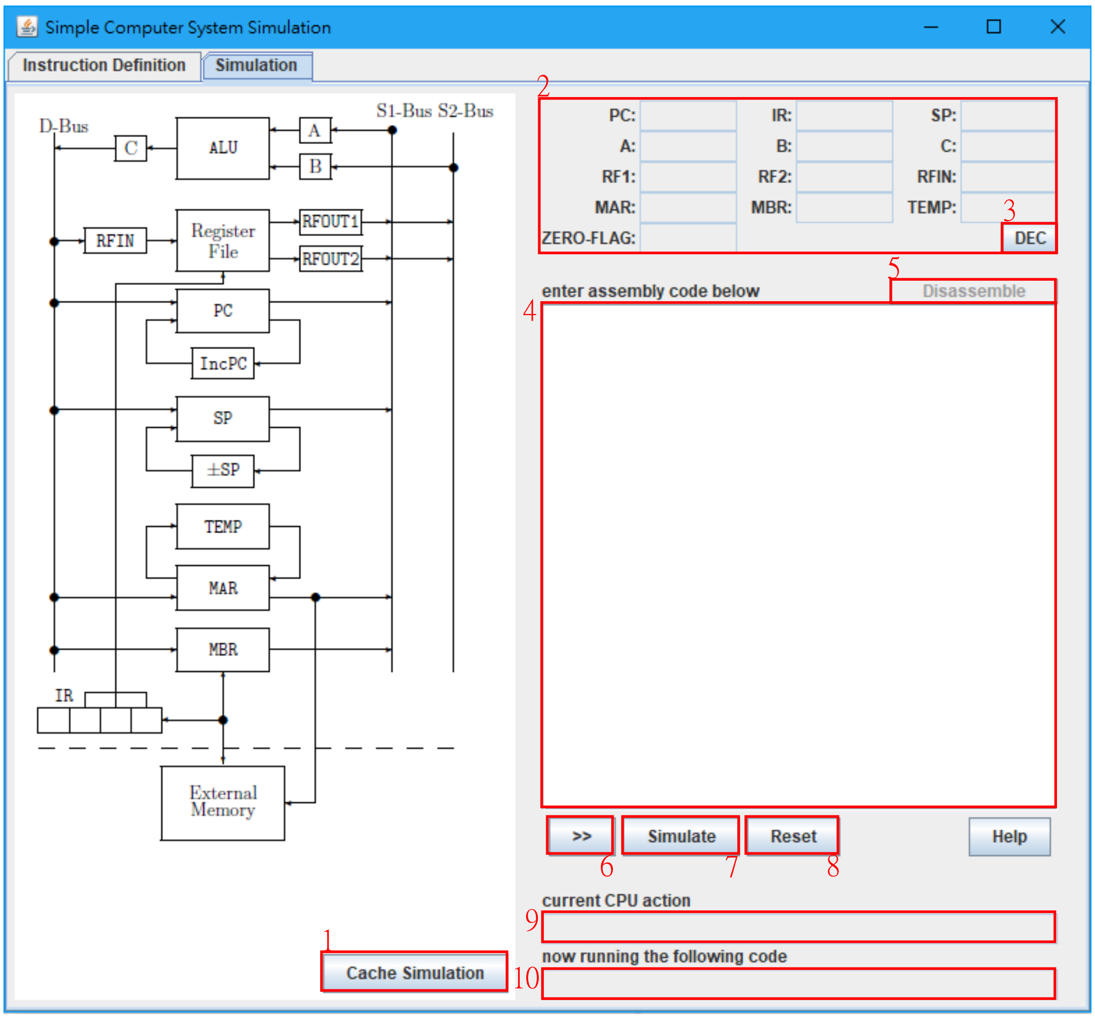

Figure 2

To simulate cache memory operations

The following lists the functionalities of different interface components as shown in figure 3 for cache memory simulation.

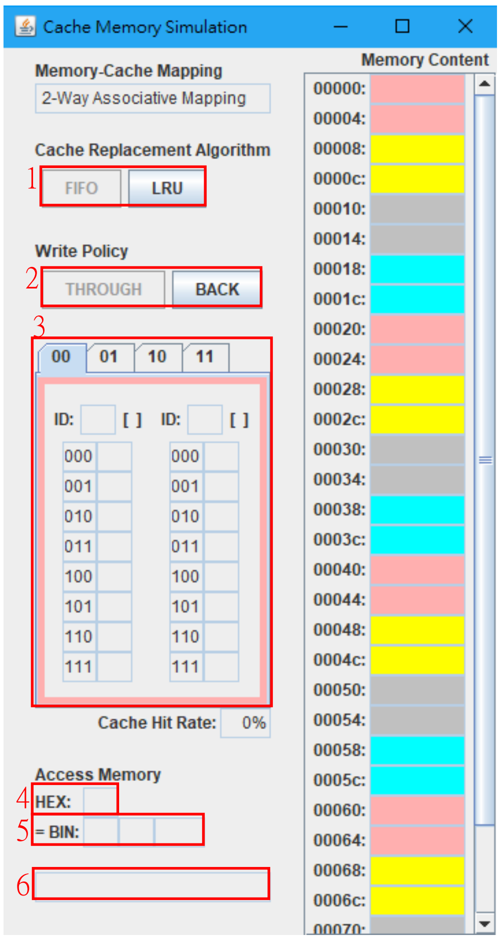
Cache Memory Simulation
Memory-Cache Mapping
2-Way Associative Mapping
Cache Replacement Algorithm
1
FIFO
LRU
Write Policy
2
THROUGH
BACK
3
00
01
10
11
ID: [ ] ID: [ ]
000
001
010
011
100
101
110
111
Cache Hit Rate: 0%
Access Memory
4
HEX:
5
= BIN:
6
Memory Content
00000: 00004: 00008: 0000c: 00010: 00014: 00018: 0001c: 00020: 00024: 00028: 0002c: 00030: 00034: 00038: 0003c: 00040: 00044: 00048: 0004c: 00050: 00054: 00058: 0005c: 00060: 00064: 00068: 0006c: 00070:

Figure 3

#	Function
1	Switch between replacement algorithms
2	Switch between write policies
3	Display of cache content, 4 cache sets each with 2 cache lines
4	Hexadecimal representation of the memory address being accessed
5	Binary representation of the memory address being accessed, for reference to the location of data in cache
6	Descriptive text field showing operation-related information

Cache simulation will run in parallel when the simulation of CPU execution starts and update the memory content as the simulation proceeds. The settings of cache replacement algorithm and write policy can only be changed before the simulation starts.

Syntax for definition txt file

Besides exporting saved instructions to an instruction definition txt file, one can also compose the txt file manually following the syntax rules below.

1. Begin an instruction definition with the mnemonic symbol of the instruction, and then a space, then a 2-digit hexadecimal operation code.

2. Starting from the next line, each line represents a CPU action entry. Begin the line with the sequence index, starting from "1", followed by a space, then either "*", "f" or "e" to indicate the addressing mode in which the action will run, then space, then the action string from the list of 22 actions, space and finally the supplementary attributes depending on the action.
   i. For bus-related actions, i.e. move_via_s1/s2/d, after the action string, enter space, source register, space, followed by destination register.
   ii. For ALU actions, after the action string, enter space, then either "true" or "false" to indicate whether the result will update zero-flag.
   iii. All other actions do not require supplementary attributes
3. For parallel actions, begin the line with the same index number while for serial actions, increment the index number by 1 every line.
4. At the end of the instruction definition after entering the last action, begin a new line and enter "---", followed by another empty line before defining another instruction.