



CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE · INDIA

**DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING**

(CS435P)
Computer Organization & Architecture

*B. Tech – Computer Science and Engineering
(Artificial Intelligence and Machine Learning)*

**School of Engineering and Technology,
CHRIST (Deemed to be University),
Kumbalagodu, Bengaluru-560 074**

April 2023



CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE • INDIA

Certificate

This is to certify that Ashvath Suresh Babu Piriya has successfully completed the record + Mini Project work for (CS435P - Computer Organization & Architecture) in partial fulfilment for the award of Bachelor of Technology in Computer Science and Engineering (Artificial Intelligence & Machine Learning) during the year 2022-2023.

HEAD OF THE DEPARTMENT

FACULTY- IN CHARGE

EXAMINER 1:

EXAMINER 2:

Name : Ashvath S.P

Register No. : 2162014

Examination Center : SoET, CHRIST (Deemed to be University)

Date of Examination :

COA LAB**Experiment – 1**

Problem statement: Write an assembly language program to perform arithmetic operations.

Algorithm:

- Step 1:** Define the Base Register Address value during the program creation.
- Step 2:** Move the first operand in the General-Purpose Register R1.
- Step 3:** Move the second operand in the General-Purpose Register R2.
- Step 4:** Perform the arithmetic operation with the values in the registers.
- Step 5:** Result will be stored in the destination register.
- Step 6:** Store the resultant value in a data memory location.
- Step 7:** Terminate the program.

Assembly Language Code:

- **Addition**

MOV #10, R01 ;*Store value of 10 in register R01*

MOV #5, R02 ;*Store value of 05 in register R02*

ADD R02, R01 ;*Add the register R01 and R02 values and store the resultant value in register R01*

STB R01, 00 ;*Store the resultant value of R01 in memory location 00*

HLT ;*Stop the simulator*

- **Subtraction**

MOV #20, R03 ;*Store value of 20 in register R03*

MOV #15, R04 ;*Store value of 15 in register R04*

SUB R04, R03 ;*Subtract the register R03 and R04 values and store the resultant value in register R03*

STB R03, 08 ;*Store the resultant value of R03 in memory location 08*

HLT ;*Stop the simulator*

- **Multiplication**

MOV #6, R05 ;*Store value of 06 in register R05*

MOV #3, R06 ;Store value of 03 in register R06

MUL R06, R05 ;Multiplicate the register R05 and R06 values and store the resultant value in register R05

STB R05, 16 ;Store the resultant value of R05 in memory location 16

HLT ;Stop the simulator

- **Division**

MOV #8, R07 ;Store value of 08 in register R07

MOV #2, R08 ;Store value of 02 in register R08

DIV R08, R07 ;Divide the register R07 and R08 values and store the resultant value in register R07

STB R07, 24 ;Store the resultant value of R07 in memory location 24

HLT ;Stop the simulator

Result:

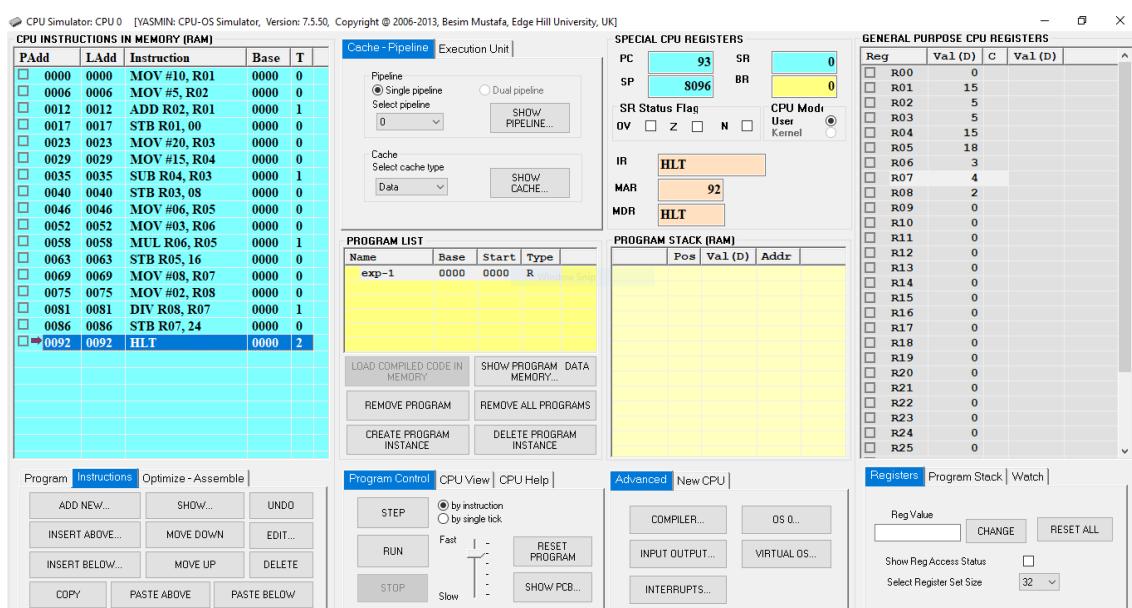


Fig. 1: CPU Simulator Window

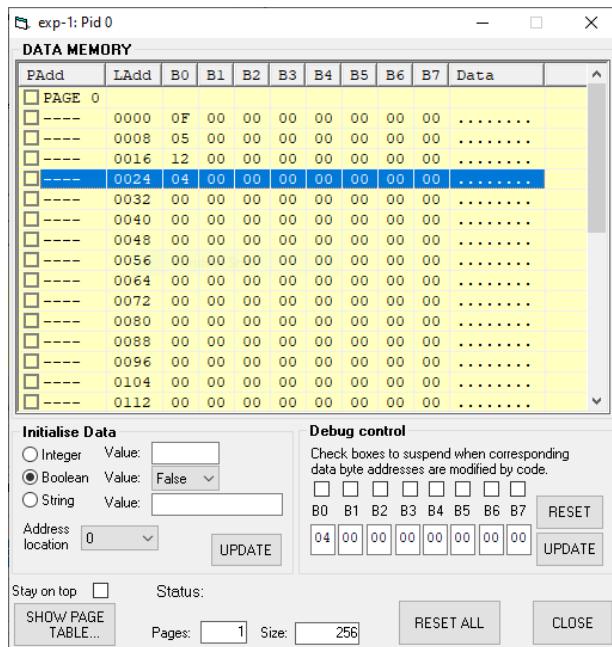


Fig. 2: Data Memory Window

Step 01 - Addition Program Starts	
PC	6
IR	MOV #10, R01
MAR	0
MDR	MOV #10, R01
R01	10
Step 02	
PC	12
IR	MOV #5, R02
MAR	6
MDR	MOV #5, R02
R01	10
R02	5
Step 03	
PC	17
IR	ADD R02, R01
MAR	12
MDR	ADD R02, R01
R01	15
R02	5
Step 04 - Addition Program Ends	
PC	23
IR	STB R01,00
MAR	0
MDR	15

R01	15
R02	5
00	0F

Step 05 - Subtraction Program Starts

PC	29
IR	MOV #20, R03
MAR	23
MDR	MOV #20, R03
R01	15
R02	5
R03	20
00	0F

Step 06

PC	35
IR	MOV #15, R04
MAR	29
MDR	MOV #15, R04
R01	15
R02	5
R03	20
R04	15
00	0F

Step 07

PC	40
IR	SUB R04, R03
MAR	29
MDR	SUB R04, R03
R01	15
R02	5
R03	5
R04	15
00	0F

Step 08 - Subtraction Program Ends

PC	46
IR	STB R03, 08
MAR	8
MDR	5
R01	15
R02	5
R03	5
R04	15
00	0F

08	05
Step 09 - Multiplication Program Starts	
PC	52
IR	MOV #06, R05
MAR	46
MDR	MOV #06, R05
R01	15
R02	5
R03	5
R04	15
R05	6
00	0F
08	05
Step 10	
PC	58
IR	MOV #03, R06
MAR	52
MDR	MOV #03, R06
R01	15
R02	5
R03	5
R04	15
R05	6
R06	3
00	0F
08	05
Step 11	
PC	63
IR	MUL R06, R05
MAR	58
MDR	MUL R06, R05
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
00	0F
08	05
Step 12 - Multiplication Program Ends	
PC	69
IR	STB R05, 16

MAR	16
MDR	18
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
00	0F
08	05
16	12

Step 13 - Division Program Starts

PC	75
IR	MOV #08, R07
MAR	69
MDR	MOV #08, R07
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
R07	8
00	0F
08	05
16	12

Step 14

PC	81
IR	MOV #02, R08
MAR	75
MDR	MOV #02, R08
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
R07	8
R08	2
00	0F
08	05
16	12

Step 15	
PC	86
IR	DIV R08, R07
MAR	81
MDR	DIV R08, R07
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
R07	4
R08	2
00	0F
08	05
16	12
Step 16 - Division Program Ends	
PC	92
IR	STB R07, 24
MAR	24
MDR	4
R01	15
R02	5
R03	5
R04	15
R05	18
R06	3
R07	4
R08	2
00	0F
08	05
16	12
24	04
Step 17 - Simulation is Terminated	
PC	93
IR	HLT
MAR	92
MDR	HLT
R01	15
R02	5
R03	5
R04	15

Date: 09-01-2023

Name: Ashvath S.P

Reg No: 2162014

R05	18
R06	3
R07	4
R08	2
00	0F
08	05
16	12
24	04

COA LAB

Experiment – 2

Problem Statement: Write an assembly language program to compute the average of two numbers.

Algorithm:

- Step 1:** Define the Base Register Address value during the program creation.
- Step 2:** Move the first operand in the General-Purpose Register R1.
- Step 3:** Move the second operand in the General-Purpose Register R2.
- Step 4:** Perform the addition operation with the values in the registers.
- Step 5:** Result will be stored in the destination register.
- Step 6:** Divide the destination register value by 2, and the result will be stored in the destination register.
- Step 7:** Store the resultant value in a data memory location.
- Step 8:** Terminate the program.

Assembly Language code:

MOV #6, R01 ;*Store value of 6 in register R01*

MOV #4, R02 ;*Store value of 4 in register R02*

ADD R01, R02 ;*Add the register R01 and R02 values and store the resultant value in register R02*

DIV #2, R02 ;*Divide register R02 by value 2 and store the resultant value in register R02*

STB R01, 00 ;*Store the resultant value of R01 in memory location 00*

HLT ;*Stop the simulator*

Result:

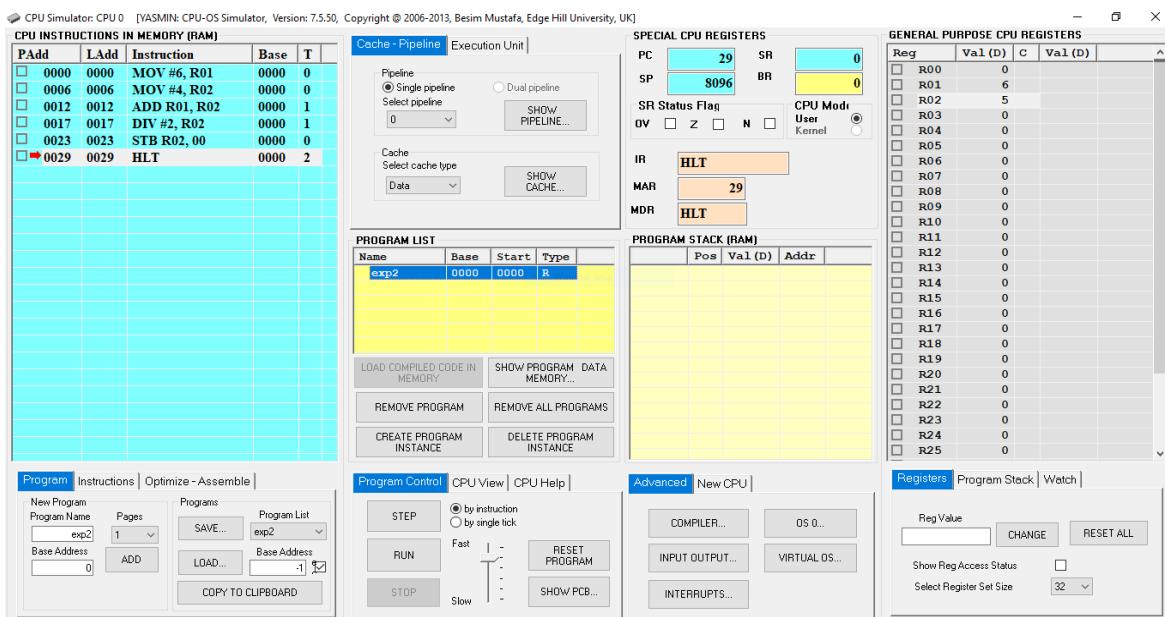


Fig. 1: CPU Simulator Window

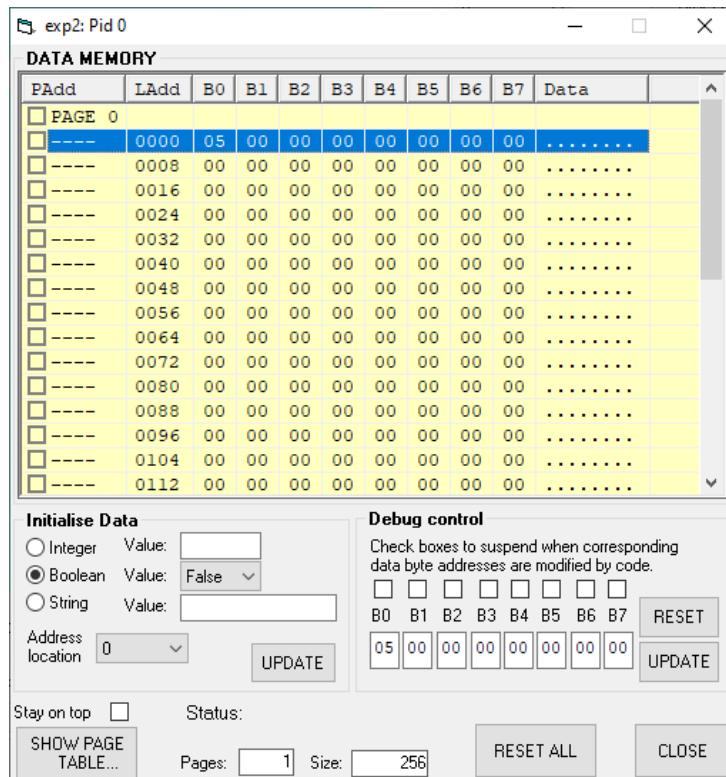


Fig. 2: Data Memory Window

Step 01	
PC	6
IR	MOV #6, R01
MAR	0
MDR	MOV #6, R01
R01	6

Step 02	
---------	--

PC	12
IR	MOV #4, R02
MAR	6
MDR	MOV #4, R02
R01	6
R02	4
Step 03	
PC	17
IR	ADD R01, R02
MAR	12
MDR	ADD R01, R02
R01	6
R02	10
Step 04	
PC	23
IR	DIV #2, R02
MAR	17
MDR	DIV #2, R02
R01	6
R02	5
00	05
Step 05	
PC	29
IR	STB R02, 00
MAR	0
MDR	5
R01	6
R02	5
00	05
Step 06	
PC	30
IR	HLT
MAR	29
MDR	HLT
R01	6
R02	5
00	05

COA LAB**Experiment – 3**

Problem Statement: Write an assembly language program to compute factorial for a given number.

Algorithm:

- Step 1:** Define the Base Register Address value during the creation of the program
- Step 2:** Move the operand to Register R1 for which you need to find out the factorial
- Step 3:** Move the Register R1 value to the R0
- Step 4:** Move the value 1 to Register R2
- Step 5:** Create a label named ‘factorial’
- Step 6:** Multiply Register R1 with register R2 and store result in R2 register
- Step 7:** Decrement Register R1 value
- Step 8:** Compare Register R1 with value 1
- Step 9:** If the Register R1 is greater than 1, jump to the ‘factorial’ label
- Step 10:** If the Register R1 is lower than or equal to 1, store the resultant factorial value in the memory location
- Step 11:** Halt the simulator

Assembly Language code:

MOV #5, R01 ;Store value of 5 in register R01

MOV R01, R00 ;Move register R01 value to R00.

MOV #1, R02 ;Store value of 1 in register R02

factorial: ;Label for factorial

MUL R01, R02 ;Multiply registers R01 to R02 and store the resultant value in register R02

DEC R01 ;Decrement register R01 value by 1

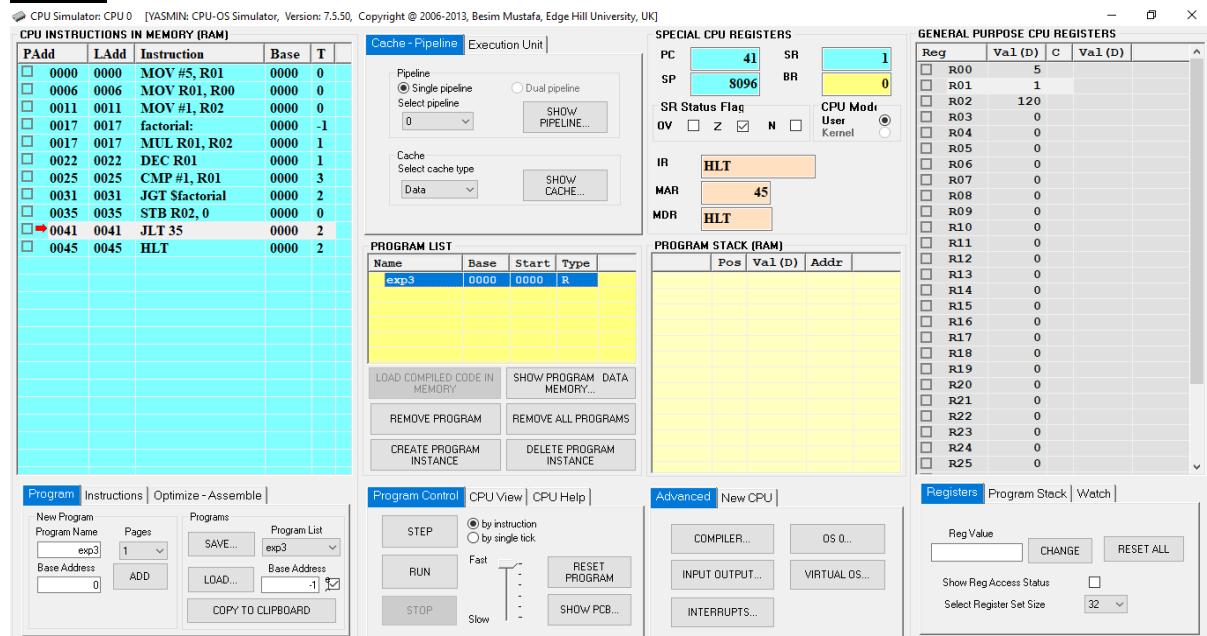
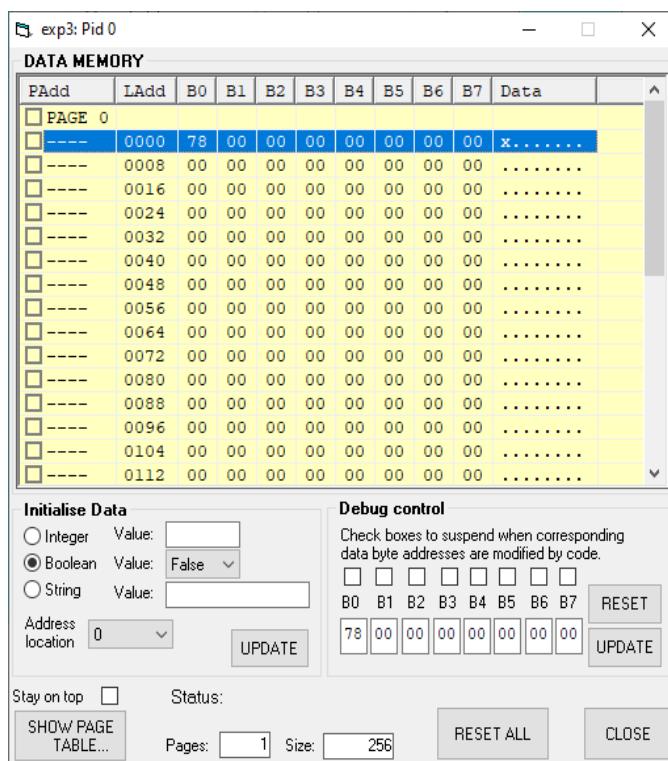
CMP #1, R01 ;Compare register R01 value by 1

JGT \$factorial ;If register R01 value is greater than 1, jump to the ‘factorial’ label

STB R02, 0 ;Store register R02 value in memory location 0

JLT 35 ;If register R01 value is less than or equal to 1, jump to the statement of Paddr 35

HLT ;Stop the simulator

Result:**Fig. 1:** CPU Simulator Window**Fig. 2:** Data Memory Window

COA LAB
Experiment – 4

Problem Statement: Write an assembly language program to check whether a given number is odd or even.

Algorithm:

1. Define the Base Register Address value during the creation of the program
2. Move the operand to the Register R1
3. Move the Register R1 value to the R0
4. Perform bitwise and operation on R1value and decimal number 1
5. Compare whether the resulting value is zero
6. If the resulting value is zero, jump to the Even label, set the Register R4 to 1 and exit
7. If the resulting value is not zero, jump to the Odd label, set the Register R5 to 1 and exit

Assembly Language code:

MOV #37, R01 ;*Store value of 37 in register R01*

MOV R01, R00 ;*Move register R01 value to R00.*

AND #1, R01 ;*AND Operation on R1 value and decimal number 1*

CMP #0, R01 ;*Compare register R01 value by 0*

JEQ \$EVEN ;*If register R01 value is equal to 0, jump to the 'EVEN' label*

JNE \$ODD ;*If the register R01 value is not equal to 0, jump to the 'ODD' label*

EVEN: ;*Label for identifying even numbers*

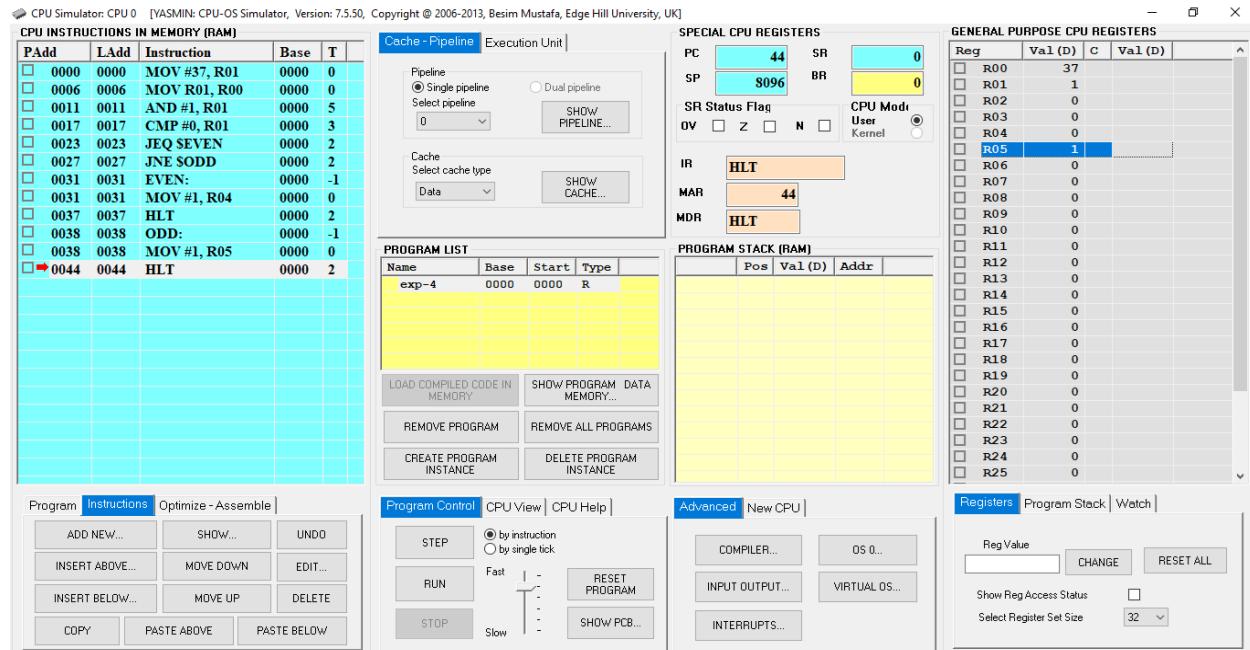
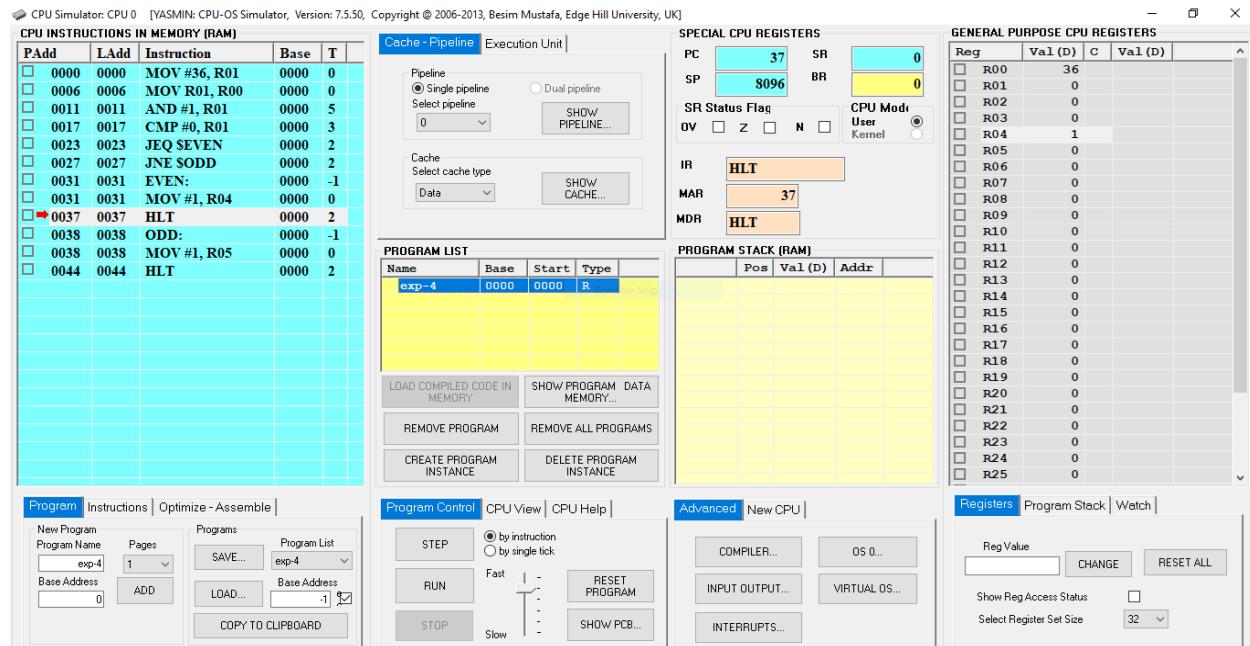
MOV #1, R04 ;*Store value of 1 in register R04*

HLT ;*Stop the simulator*

EVEN: ;*Label for identifying odd numbers*

MOV #1, R05 ;*Store value of 1 in register R05*

HLT ;*Stop the simulator*

Result:**Case 1: Odd****Fig.1: CPU Simulator Window****Case 2: Even****Fig.2: CPU Simulator Window**

COA LAB
Experiment - 5

Write an assembly language for the following conditional statements:

1. If R02 is greater than R01, R03 is set to 8. (Use R01 as the first operand and R02 as the second operand).
2. If R01 = 0, R03 is set to 5, else R03 is set to R01 plus 1.
3. A loop that repeats 5 times where R02 is incremented by 2 every time the loop repeats.
4. A loop that repeats while R04 is > 0. Set the initial value of R04 to 8.
5. A loop that repeats until R05 is > R09. Set the initial values of R05 to 0 and R09 to 12.

Program no. 1

Algorithm:

1. Load the value 15 into R01
2. Load the value 5 into R02
3. Compare R01 and R02 with the CMP instruction
4. If R02 is greater than R01, jump to the ADD8 label
5. If R02 is not greater than R01, halt the program with the HLT instruction
6. At the ADD8 label, load the value 8 into R03
7. Halt the program with the HLT instruction

Assembly Language code:

```
MOV #15, R01 ;Load the value 15 into R01
MOV #5, R02 ;Load the value 5 into R02
CMP R01, R02 ;Compare R01 and R02
JGT ADD8 ;If R02 is greater than R01, jump to the ADD8 label
HLT ;If R02 is not greater than R01, halt the program
ADD8: ;Label for the ADD8 instruction
MOV #8, R03 ;Load the value 8 into R03
HLT ;Halt the program
```

Result:

Case 1: R02 > R01

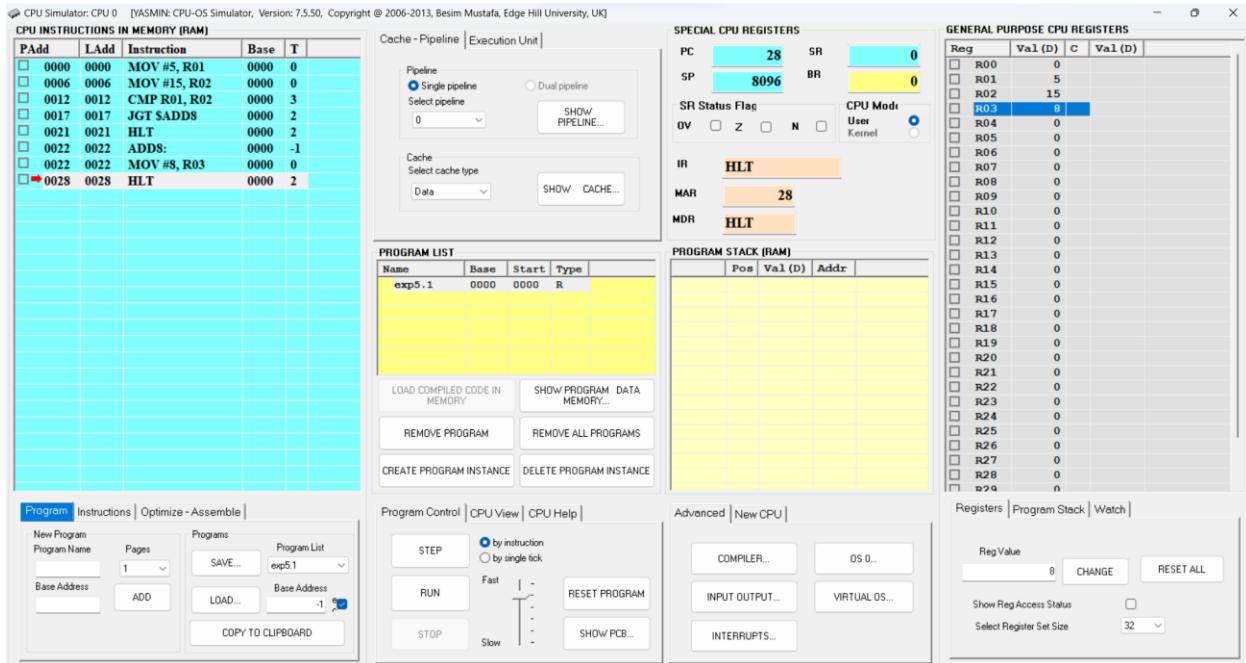


Fig.1: CPU Simulator Window

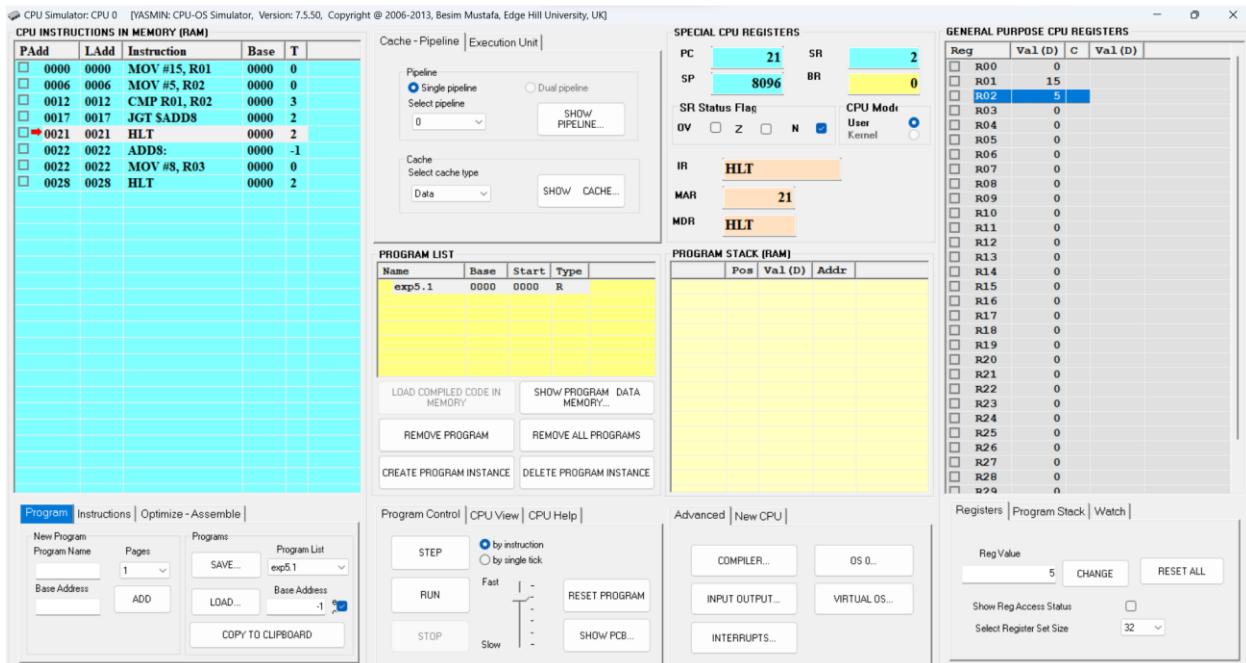
Case 2: R02 < R01

Fig.2: CPU Simulator Window

Program no. 2**Algorithm:**

1. Initialize R01 to 0
2. Compare R01 with 0

3. If R01 equals 0, go to IF statement
4. If R01 does not equal 0, go to ELSE statement
5. At IF statement, set R03 to 5
6. At ELSE statement, add 1 to R01 and set R03 to R01
7. Halt the program.

Assembly Language code:

```

MOV #0, R01 ;Initialize R01 to 0
CMP #0, R01 ;Compare R01 with 0
JEQ $IF ;If R01 equals 0, go to IF statement
JNE $ELSE ;If R01 does not equal 0, go to ELSE statement
$IF: ;At IF statement, set R03 to 5
MOV #5, R03
HLT ;Halt the program
$ELSE: ;At ELSE statement, add 1 to R01 and set R03 to R01
ADD #1, R01
MOV R01, R03
HLT ;Halt the program

```

Result:

Case 1: R01 = 0

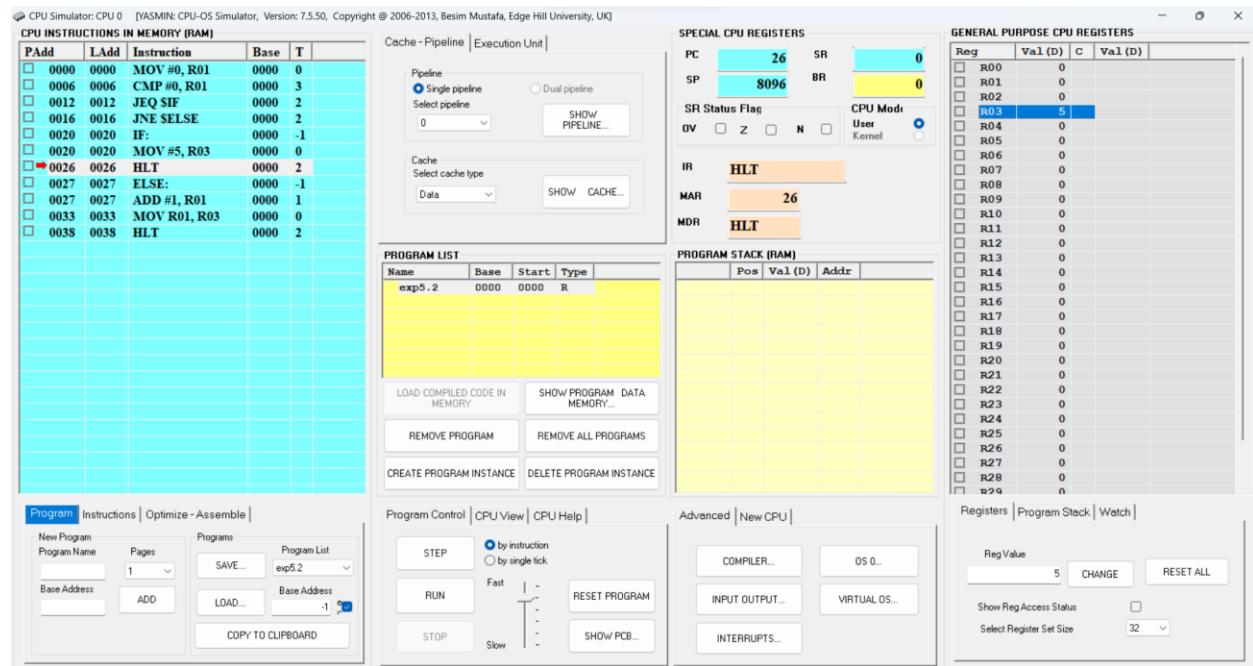


Fig.3: CPU Simulator Window

Case 2: R01 != 0

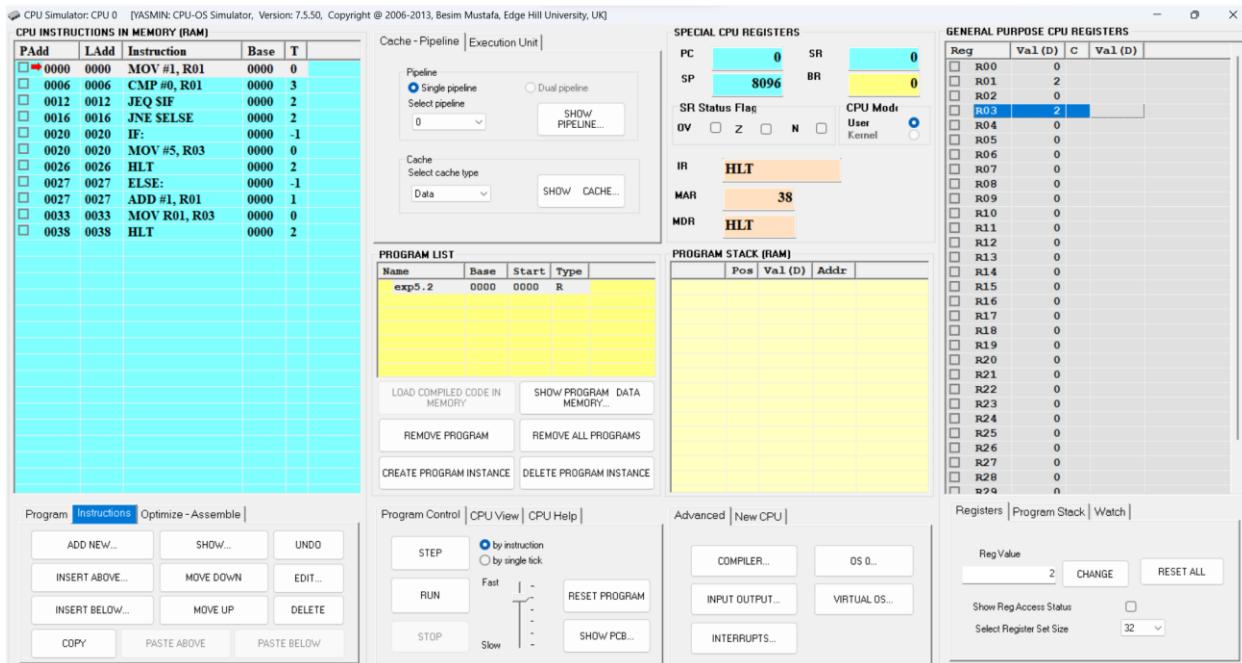


Fig.4: CPU Simulator Window

Program no. 3

Algorithm:

1. Initialize the value of R01 to 1.
2. Initialize the value of R02 to 5.
3. Repeat the following steps until R01 is equal to 5:
 - a. Increment R02 by 2.
 - b. Increment R01 by 1.
4. Halt the program.

Assembly Language code:

```

MOV #1, R01 ;move the value 1 into register R01
MOV #5, R02 ;move the value 5 into register R02
LOOP: ;label the start of the loop as "LOOP"
ADD #2, R02 ;add the value 2 to register R02
ADD #1, R01 ;add the value 1 to register R01
CMP #5, R01 ;compare the value in register R01 to 5
JNE $LOOP ;jump to "LOOP" if R01 is not equal to 5
HLT ;halt the program

```

Result:

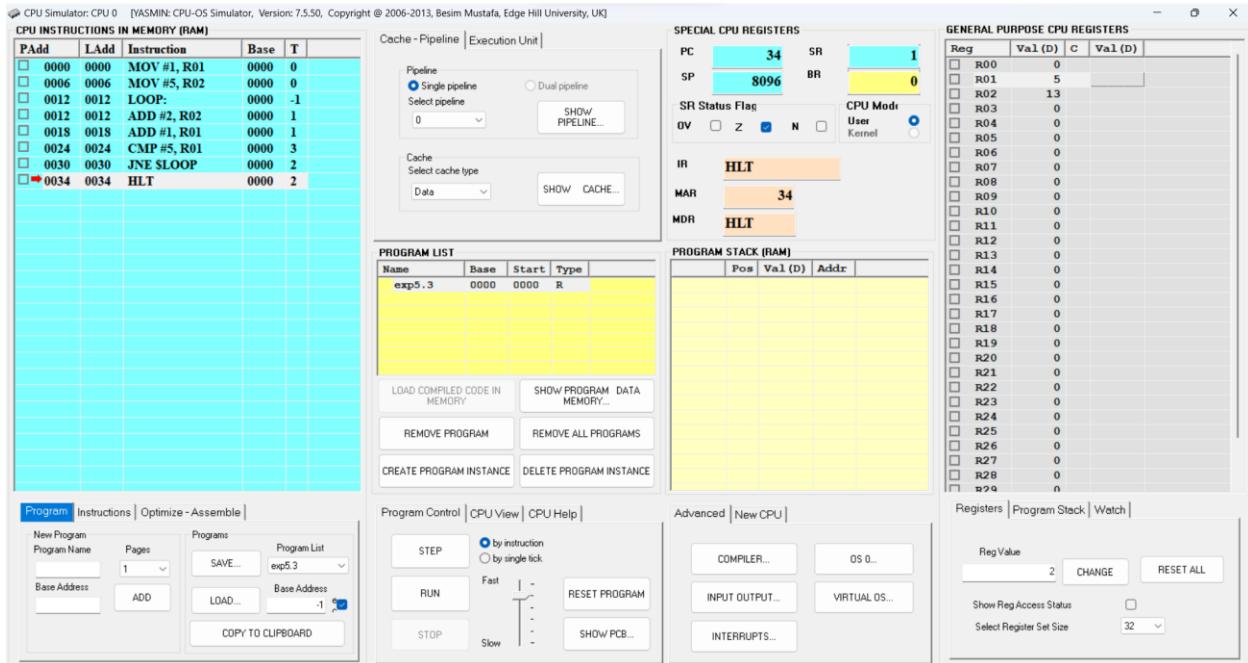


Fig.5: CPU Simulator Window

Program no. 4

Algorithm:

1. Initialize R04 to 8.
2. Start the loop: "LOOP"
3. Subtract 1 from R04.
4. Compare R04 to 0.
5. If R04 is not equal to 0, go back to the loop.
6. If R04 is equal to 0, halt the program.

Assembly Language code:

```

MOV #8, R04 ;Move value 8 into register R04
LOOP: ;Label for the start of the loop
SUB #1, R04 ;Subtract value 1 from the contents of R04
CMP #0, R04 ;Compare the contents of R04 with value 0
JNE LOOP ;If R04 is not equal to 0, jump back to the label LOOP
HLT ;Halt the program execution.

```

Result:

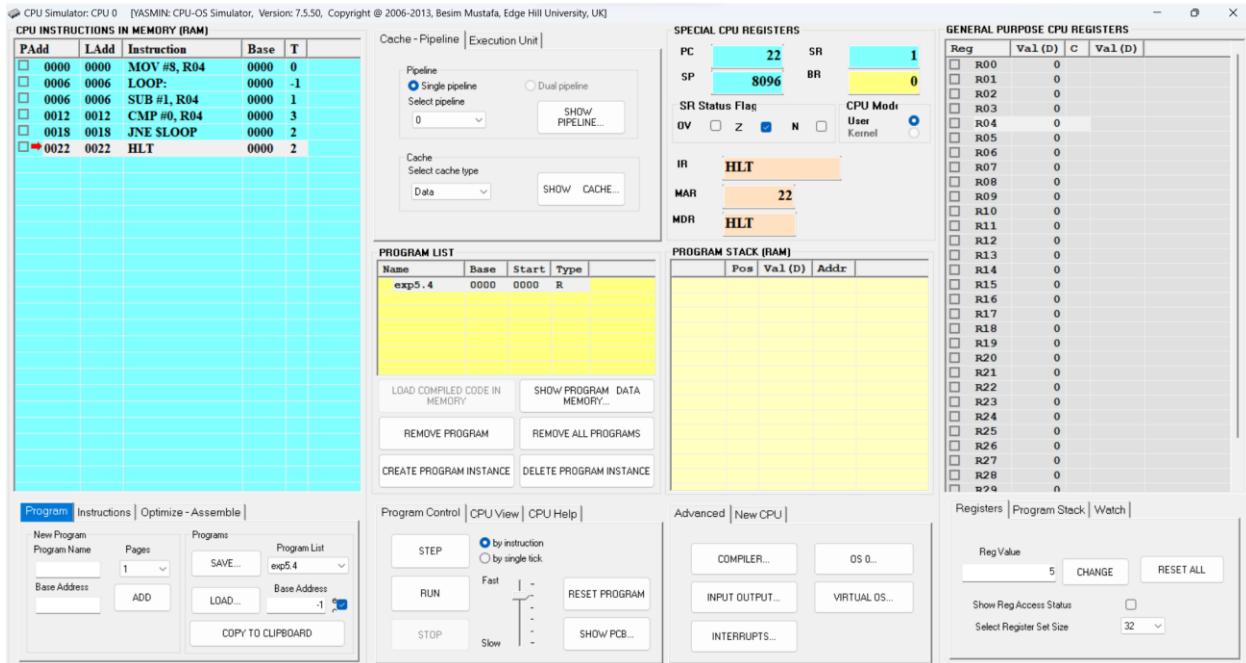


Fig.6: CPU Simulator Window

Program no. 5

Algorithm:

1. Initialize R05 to 0 and R09 to 12.
2. Start the loop with the label "LOOP".
3. Decrement R09 by 1.
4. Compare R05 and R09. If R05 is greater than R09, go to step 7.
5. If R05 is not greater than R09, repeat from step 3.
6. End the loop.
7. Halt the program.

Assembly Language code:

```

MOV #0, R05 ;Initialize R05 to 0
MOV #12, R09 ;Initialize R09 to 12
LOOP: ;Start the loop
SUB #1, R09 ;Decrement R09 by 1
CMP R05, R09 ;Compare R05 and R09
JGT $LOOP ;If R05 is greater than R09, jump to the end of the loop
HLT ;Halt the program

```

Result:

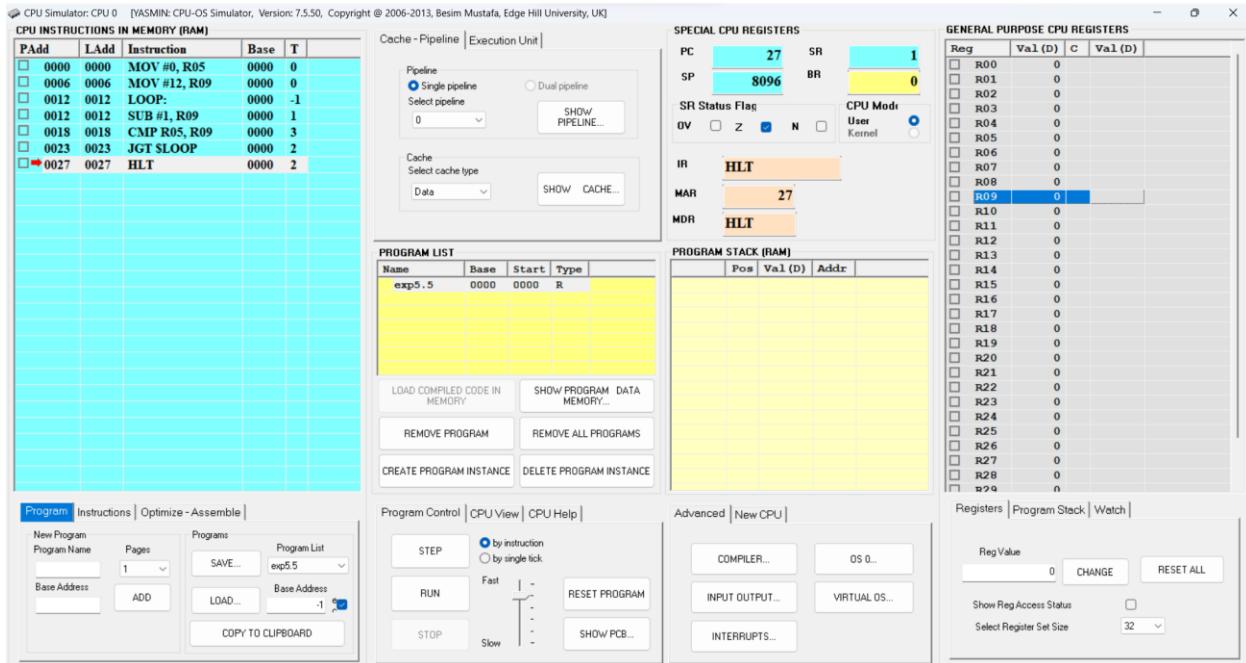


Fig.7: CPU Simulator Window

COA LAB
Experiment – 6

Write an assembly language program for the following problem statements:

1. A routine that pushes numbers 6 and 4 on top of the stack, then pops the two numbers one by one from stack, add them and pushes the result back to the top of stack.
2. First loop places 15 numbers from 1-15 on top of stack using push instruction. In the second loop, use the pop instruction to pop two numbers from the top of the stack, add them, and push the result back to the stack. Repeat the second loop until only one number is left on top of the stack.

Program no. 1

Algorithm:

1. Mark the start of the frame
2. Create a label ‘pushpopadd’ to represent the routine name.
3. Call routine ‘pushpopadd’.
4. Place the halt instruction below the call instruction and move the label ‘pushpopadd’ below the halt instruction.
5. Inside the label/routine ‘pushpopadd’, push numbers 6 and 4 on top of stack, then pop the two numbers one by one from stack, add them and push the result back to the top of stack.
6. Pop to any general-purpose register to get the return address on top of the stack
7. Returns to Halt instruction and exit.

Assembly Language code:

MSF ;Main start function

CAL \$pushpopadd ;Call the function pushpopadd

HLT ;Halt the program execution

pushpopadd: ;Define the function pushpopadd

PSH #6 ;Push the value 6 onto the stack

PSH #4 ;Push the value 4 onto the stack

POP R01 ;Pop the top value from the stack into register R01

POP R02 ;Pop the second value from the stack into register R02

ADD R01, R02 ;Add the values in R01 and R02 and store the result in R02

PSH R02 ;Push the result onto the stack

POP R03 ;Pop the result from the stack into register R03

RET ;Return from the function

Result:

PROGRAM STACK (RAM)				
	Pos	Val (D)	Addr	
TOS->	3	4	0010	
	2	6	0006	
	1	5	0000	
BOS->	0	-1	0000	

Fig-6.1.a: Program Stack window after pushing all the values

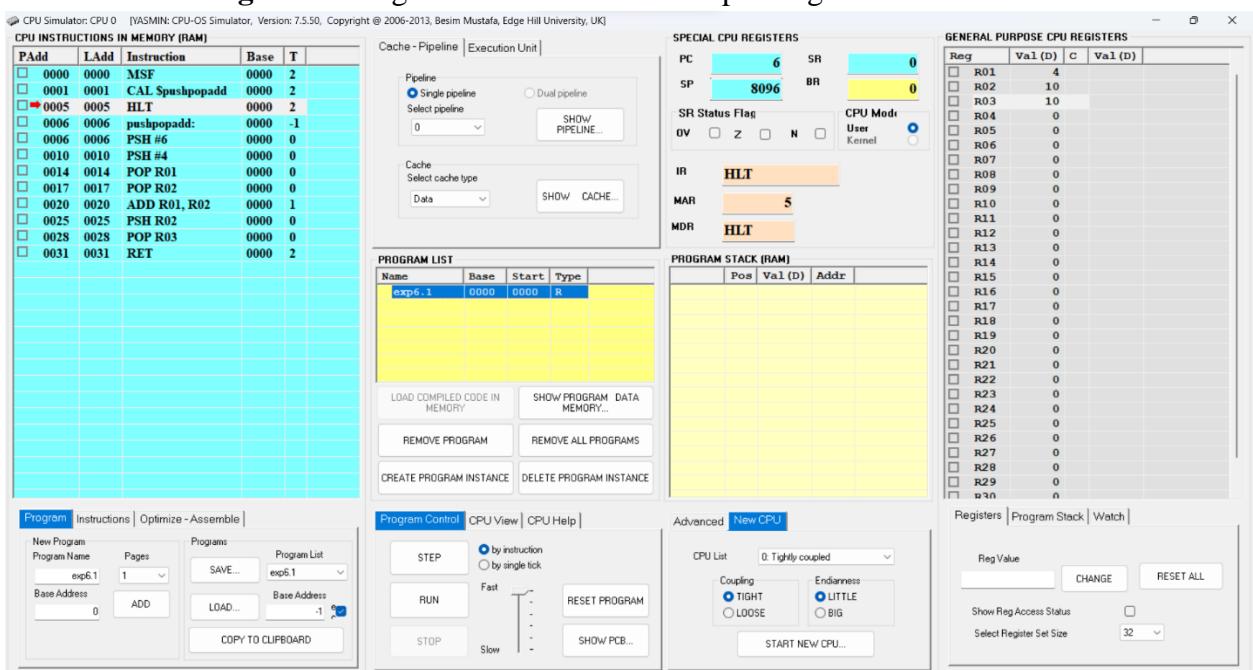


Fig-6.1.b: CPU Simulator Window

Program no. 2

Algorithm:

1. Initialize register R01 to 1 to hold the value of the first number to be pushed onto the stack.
2. Enter a loop to push 15 numbers onto the stack using the PUSH instruction, incrementing R01 each time to hold the value of the next number.

3. After the first loop completes, enter a second loop to pop two numbers from the top of the stack, add them, and push the result back to the stack.
4. Repeat the second loop until only one number is left on top of the stack.
5. The final result will be the value left on top of the stack.
6. Halt the program after the second loop completes.

Assembly Language code:

```
MOV #1, R01 ;Initialize register R01 to 1
MSF ;Main start function
CAL $pushpoploop ;Call the function pushpoploop
HLT ;Halt the program execution
pushpoploop: ;Define the function pushpoploop
PSH R01 ;Push the value of register R01 onto the stack
INC R01 ;Incrementing value of R01 by 1
CMP #15, R01 ;Compare value of R01 with 15
JLE $pushpoploop ;Jump back to pushpoploop if R01 <= 15
addloop ;Define the function addloop
POP R01 ;Pop the top value from the stack into register R01
POP R02 ;Pop the second value from the stack into register R02
ADD R01, R02 ;Add the values in R01 and R02 and store the result in R02
PSH R02 ;Push the value of register R02 onto the stack
CMP #120, R02 ;Compare value of R02 with 120
JNE $addloop ;Jump back to addloop if the result is not equal to 120
POP R03 ;Pop the final result from the top of the stack
RET ;Return from the subroutine
```

Result:

PROGRAM STACK (RAM)			
	Pos	Val (D)	Addr
□ TOS->	16	15	0012
□	15	14	0012
□	14	13	0012
□	13	12	0012
□	12	11	0012
□	11	10	0012
□	10	9	0012
□	9	8	0012
□	8	7	0012
□	7	6	0012
□	6	5	0012
□	5	4	0012
□	4	3	0012
□	3	2	0012
□	2	1	0012
□	1	11	0006
□ BOS->	0	0	0006

Fig-6.2.a: Program Stack window before exiting the first loop

PROGRAM STACK (RAM)			
	Pos	Val (D)	Addr
□ TOS->	2	120	0039
□	1	11	0006
□ BOS->	0	0	0006

Fig-6.2.b: Program Stack window before exiting the second loop

Date: 27-02-2023

Name: Ashvath S.P

Reg No: 2162014

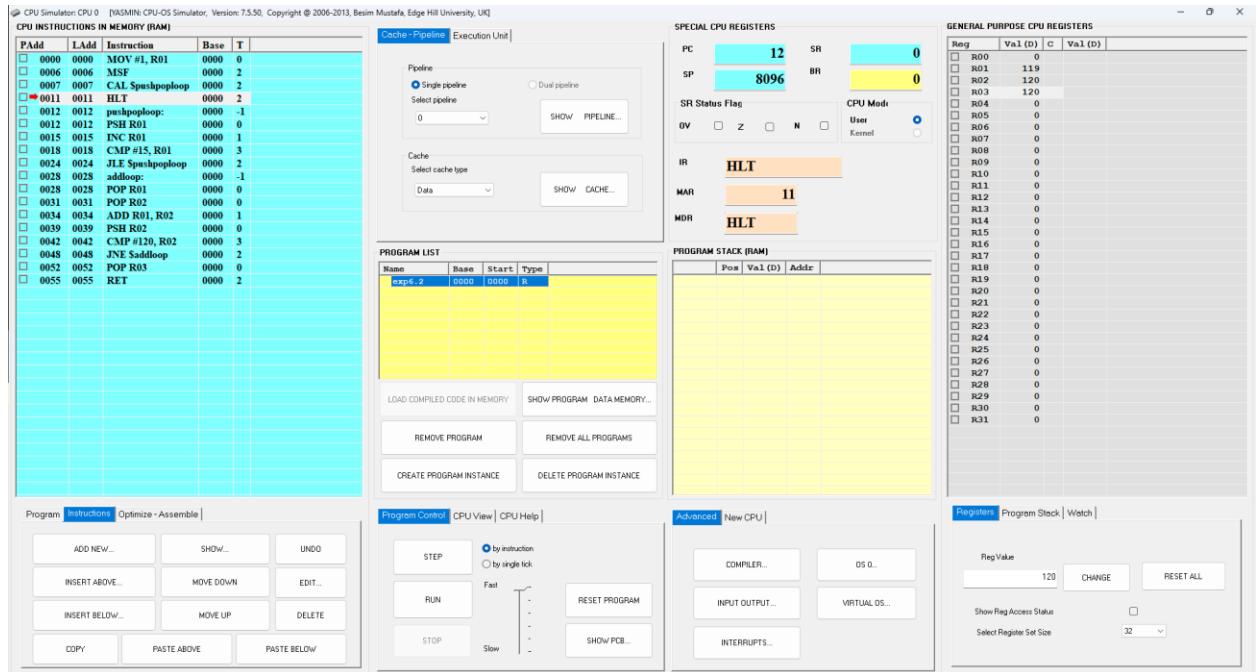


Fig-6.2.c: CPU Simulator Window

COA LAB
Experiment – 7

Write a program in Assembly Language to perform Linear Search, Binary Search.

Program no. 7.1 (Linear Search)

Algorithm:

1. Set the value of the key to be searched in register R00.
2. Initialize the values of registers R01 to R05 with the array of values to be searched.
3. Perform a linear search by iterating through the array of values and comparing each value with the key using the CMP instruction.
4. If the value in any of the registers R01 to R05 matches the key, jump to the "keyfound" label and set the value of register R10 to 1.
5. The "keyfound" label sets the value of register R10 to 1 and halts the program.

Assembly Language code:

```
MOV #15, R01 ;Store value of 15 in register R01
MOV #7, R02 ;Store value of 7 in register R02
MOV #11, R03 ;Store value of 11 in register R03
MOV #12, R04 ;Store value of 12 in register R04
MOV #9, R05 ;Store value of 9 in register R05
MOV #7, R00 ;Store value of 7 in register R00
CMP R00, R01 ;Compare value of register R01 with value of register R00
JEQ $keyfound ;If register R01 value is equal to R00, jump to the 'keyfound' label
CMP R00, R02 ;Compare value of register R02 with value of register R00
JEQ $keyfound ;If register R02 value is equal to R00, jump to the 'keyfound' label
CMP R00, R03 ;Compare value of register R03 with value of register R00
JEQ $keyfound ;If register R03 value is equal to R00, jump to the 'keyfound' label
CMP R00, R04 ;Compare value of register R04 with value of register R00
JEQ $keyfound ;If register R04 value is equal to R00, jump to the 'keyfound' label
CMP R00, R05 ;Compare value of register R05 with value of register R00
JEQ $keyfound ;If register R05 value is equal to R00, jump to the 'keyfound' label
HLT ;Halts the simulator
keyfound: ;Label for identifying the key value
MOV #1, R10 ;Store value of 1 in register R10
HLT ;Halts the simulator
```

Result:

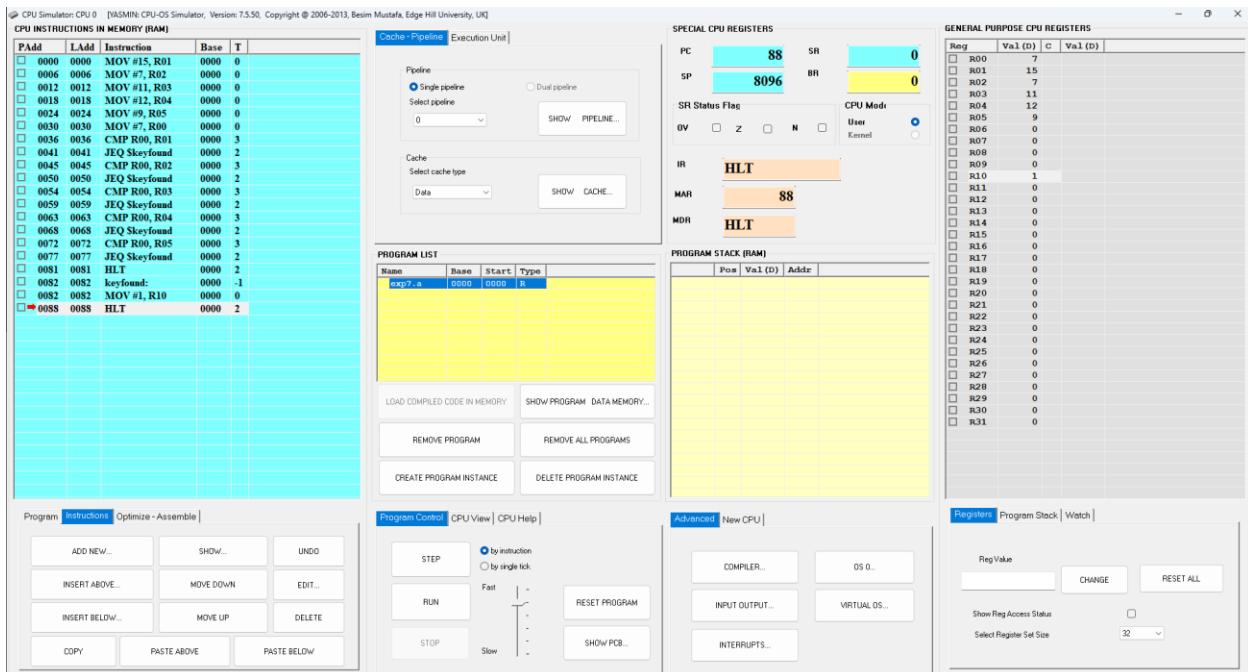


Fig7.1.a: CPU Simulator Window (key found)

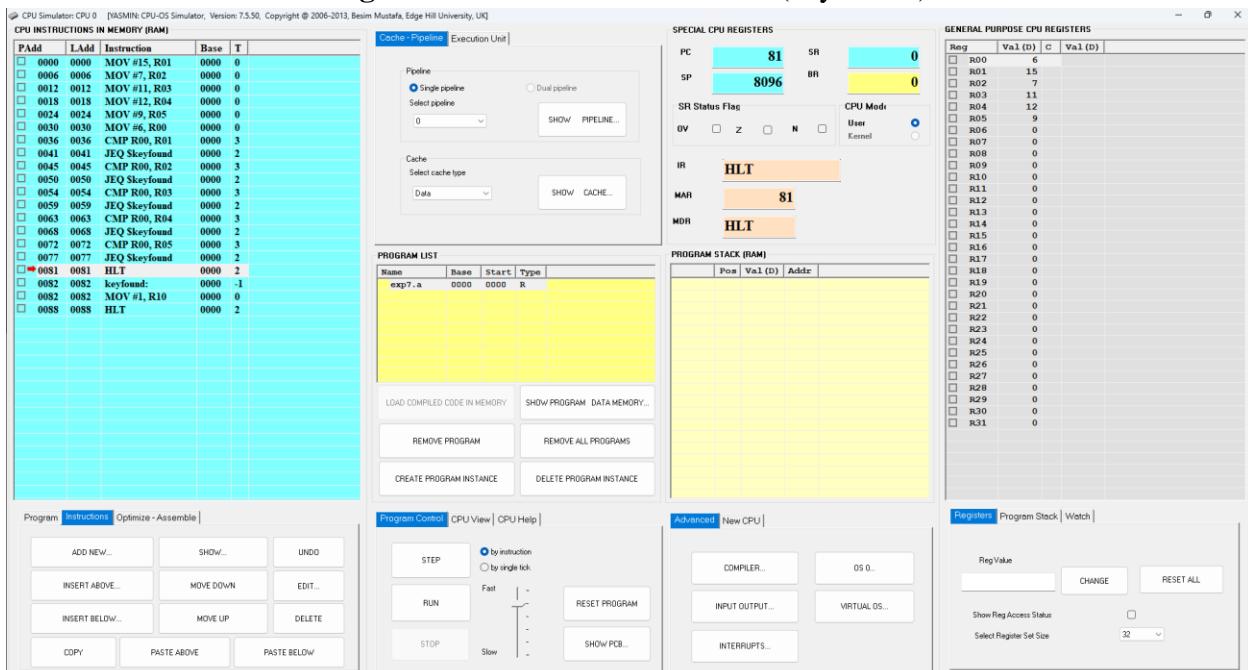


Fig7.1.b: CPU Simulator Window (key not found)

Program no. 7.2 (Binary Search)

Algorithm:

1. Set the value to be searched in a register (R00).
2. Initialize the array of values to be searched in ascending order (e.g., in registers R01 to R05).
3. Initialize the lower bound of the search range (R06) to 0 and the upper bound (R07) to the index of the last element in the array.
4. Calculate the initial midpoint as the average of the lower and upper bounds.
5. Enter a loop to perform the binary search:
 - a. Compare the value at the current midpoint with the key (R00) using CMP.
 - b. If the midpoint value is greater than the key, set the new upper bound to the midpoint - 1 and recalculate the midpoint.
 - c. If the midpoint value is less than the key, set the new lower bound to the midpoint + 1 and recalculate the midpoint.
 - d. If the midpoint value is equal to the key, set the value of a register (e.g., R10) to 1 and halt the program.
6. Continue the loop until either the key is found or the search range is exhausted (i.e., lower bound > upper bound).
7. If the key is not found, the program will halt without setting the value of the register to 1.

Assembly Language code:

```

; Move constants into registers
MOV #3, R01 ; Move 3 into register R01
MOV #5, R02 ; Move 5 into register R02
MOV #7, R03 ; Move 7 into register R03
MOV #10, R04 ; Move 10 into register R04
MOV #12, R05 ; Move 12 into register R05
MOV #15, R06 ; Move 15 into register R06
MOV #17, R07 ; Move 17 into register R07
MOV #12, R00 ; Move 12 into register R00

; Compare values and jump to appropriate labels
CMP R04, R00 ; Compare value in register R04 with value in register R00
JEQ $FOUND ; Jump to label $FOUND if they are equal
JGT $GREATER1 ; Jump to label $GREATER1 if R04 is greater than R00
JMP $LESSER1 ; Jump to label $LESSER1 if R04 is less than R00

```

GREATER1

```

CMP R06, R00 ; Compare value in register R06 with value in register R00
JEQ $FOUND ; Jump to label $FOUND if they are equal
JGT $GREATER2 ; Jump to label $GREATER2 if R06 is greater than R00

```

JMP \$LESSER2 ; *Jump to label \$LESSER2 if R06 is less than R00*

LESSER1

CMP R02, R00 ; *Compare value in register R02 with value in register R00*

JEQ \$FOUND ; *Jump to label \$FOUND if they are equal*

JGT \$GREATER3 ; *Jump to label \$GREATER3 if R02 is greater than R00*

JMP \$LESSER3 ; *Jump to label \$LESSER3 if R02 is less than R00*

GREATER2

CMP R07, R00 ; *Compare value in register R07 with value in register R00*

JEQ \$FOUND ; *Jump to label \$FOUND if they are equal*

JMP \$NOTFOUND ; *Jump to label \$NOTFOUND*

LESSER2

CMP R00, R05 ; *Compare value in register R00 with value in register R05*

JEQ \$FOUND ; *Jump to label \$FOUND if they are equal*

JMP \$NOTFOUND ; *Jump to label \$NOTFOUND*

GREATER3

CMP R00, R03 ; *Compare value in register R00 with value in register R03*

JEQ \$FOUND ; *Jump to label \$FOUND if they are equal*

JMP \$NOTFOUND ; *Jump to label \$NOTFOUND*

LESSER3

CMP R00, R01 ; *Compare value in register R00 with value in register R01*

JEQ \$FOUND ; *Jump to label \$FOUND if they are equal*

JMP \$NOTFOUND ; *Jump to label \$NOTFOUND*

FOUND

MOV #1, R10 ; *Move 1 into register R10*

HLT ; *Halt program*

NOTFOUND

MOV #1, R11 ; *Move 1 into register R11*

HLT ; *Halt program*

Result:

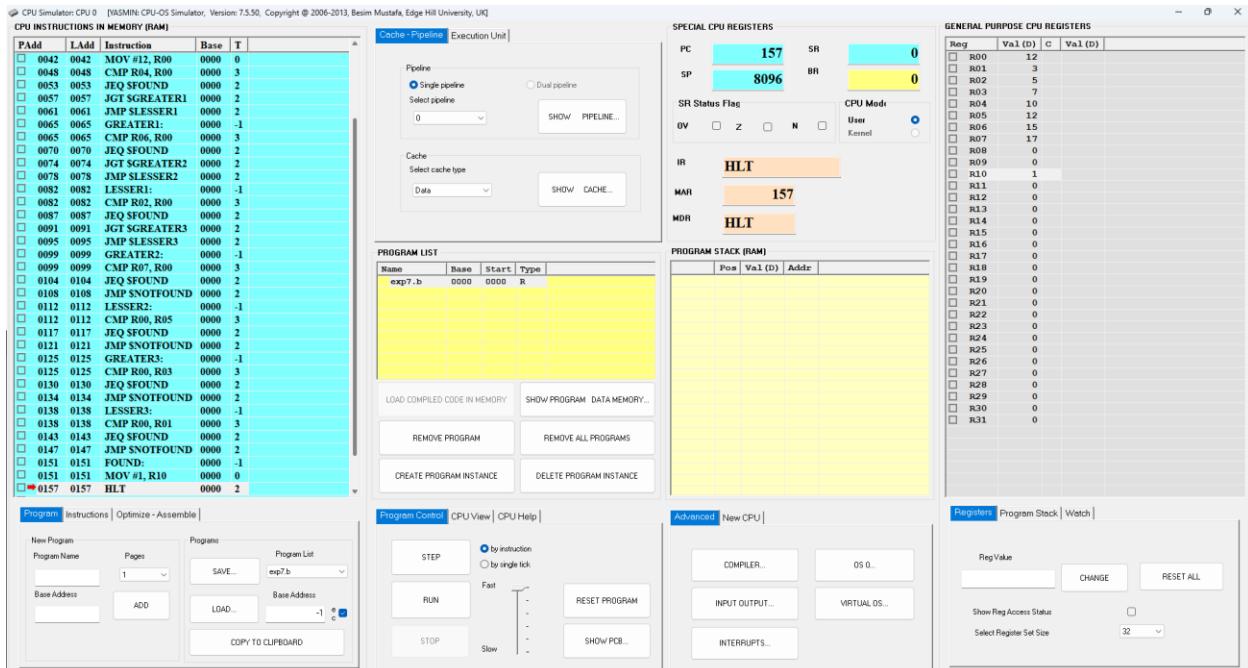


Fig7.2.a: CPU Simulator Window (key found)

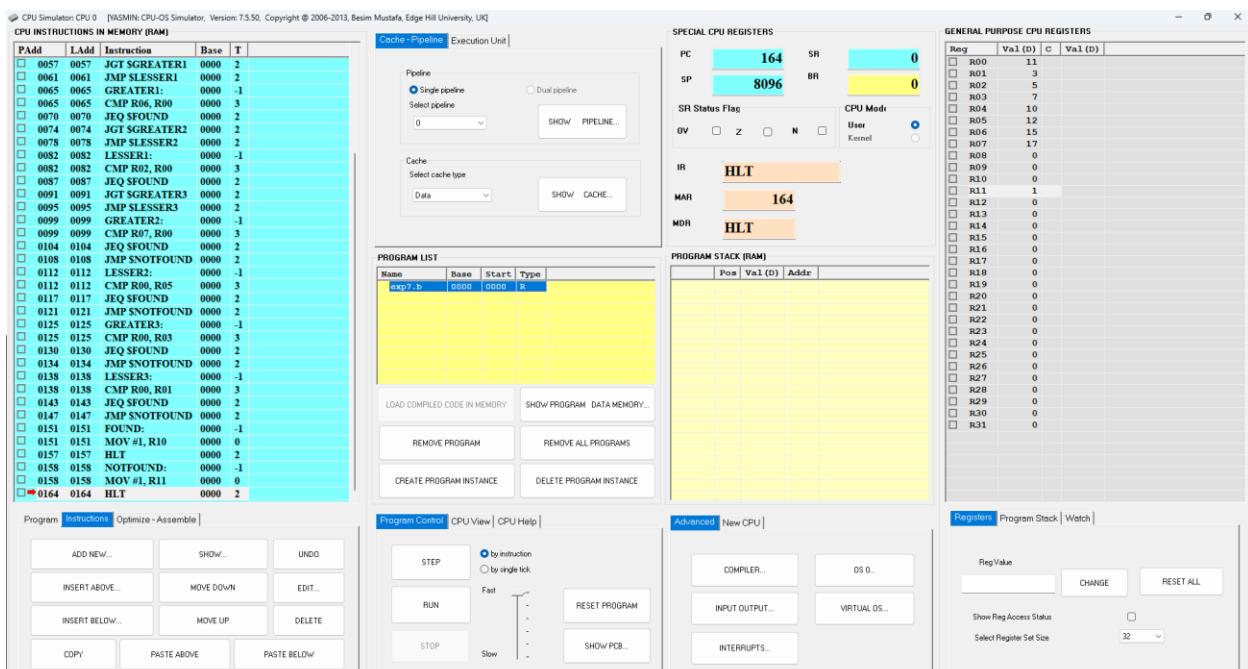


Fig7.2.b: CPU Simulator Window (key not found)

COA LAB
Experiment – 8

Write a program in Assembly Language to perform Bubble Sort.

Algorithm:

1. Load the first integer into R01, the second into R02, and so on, up to R05.
2. Set R00 to 5, the number of integers in the array.
3. Call the MSF subroutine.
4. In the MSF subroutine, decrement R00 and jump to CMP1 if it is zero.
5. Compare R01 and R02. If R01 is less than R02, jump to CMP1. Otherwise, jump to CMP2.
6. In CMP1, swap R01 and R02 and jump to JBCCMP2.
7. In CMP2, compare R02 and R03. If R02 is less than R03, jump to CMP2. Otherwise, jump to CMP3.
8. In CMP3, swap R03 and R04 and jump to JBCCMP4.
9. In CMP4, compare R04 and R05. If R04 is less than R05, jump to CMP4. Otherwise, jump to BUBBLE.
10. Jump back to the MSF subroutine.
11. When R00 reaches 0, jump to EXIT.
12. The HLT instruction at the end of the program halts execution.

Assembly Language Code:

```

MSF      ; Jump to the main subroutine
CAL $LOAD ; Call the $LOAD subroutine
MOV #5, R00 ; Move the value 5 to R00, the counter for the bubble sort loop
MSF      ; Call the MSF subroutine to start the bubble sort loop
CAL $BUBBLE ; Call the $BUBBLE subroutine to compare and swap adjacent values
in the array

```

```

LOAD    ; Load the array into registers R01-R05
MOV #15, R01 ; Move the value 15 to R01, the first integer in the array
MOV #33, R02 ; Move the value 33 to R02, the second integer in the array
MOV #27, R03 ; Move the value 27 to R03, the third integer in the array
MOV #18, R04 ; Move the value 18 to R04, the fourth integer in the array
MOV #10, R05 ; Move the value 10 to R05, the fifth integer in the array
RET     ; Return from the main subroutine

```

```

BUBBLE  ; Start the $BUBBLE subroutine
DEC R00  ; Decrement the counter R00 for the bubble sort loop
JBCCMP1 ; Jump to CMP1 if R00 is zero

```

CMP R01, R02 ; Compare R01 and R02
JLT \$CMP1 ; Jump to CMP1 if R01 is less than R02
JBCCMP2 ; Jump to CMP2 if R01 is greater than or equal to R02
CMP R02, R03 ; Compare R02 and R03
JLT \$CMP2 ; Jump to CMP2 if R02 is less than R03
JBCCMP3 ; Jump to CMP3 if R02 is greater than or equal to R03
CMP R03, R04 ; Compare R03 and R04
JLT \$CMP3 ; Jump to CMP3 if R03 is less than R04
JBCCMP4 ; Jump to CMP4 if R03 is greater than or equal to R04
CMP R04, R05 ; Compare R04 and R05
JLT \$CMP4 ; Jump to CMP4 if R04 is less than R05
CMP #0, R00 ; Compare R00 to zero
JEQ \$EXIT ; Jump to EXIT if R00 is zero
JNE \$BUBBLE ; Jump to BUBBLE if R00 is not zero
RET ; Return from the \$BUBBLE subroutine

CMP1 ; Start the CMP1 subroutine
SWP R01, R02 ; Swap the values in R01 and R02
JMP \$JBCCMP2 ; Jump back to JBCCMP2 to continue comparing and swapping adjacent values

CMP2 ; Start the CMP2 subroutine
SWP R02, R03 ; Swap the values in R02 and R03
JMP \$JBCCMP3 ; Jump back to JBCCMP3 to continue comparing and swapping adjacent values

CMP3 ; Start the CMP3 subroutine
SWP R03, R04 ; Swap the values in R03 and R04
JMP \$JBCCMP4 ; Jump back to JBCCMP4 to continue comparing and swapping adjacent values

CMP4 ; Start the CMP4 subroutine
SWP R04, R05 ; Swap the values in R04 and R05
JMP \$BUBBLE ; Jump back to BUBBLE to continue comparing and swapping adjacent values

EXIT ; Start the EXIT subroutine
HLT ; Halt the program

Result:

Date: 03-04-2023

Name: Ashvath S.P

Reg No: 2162014

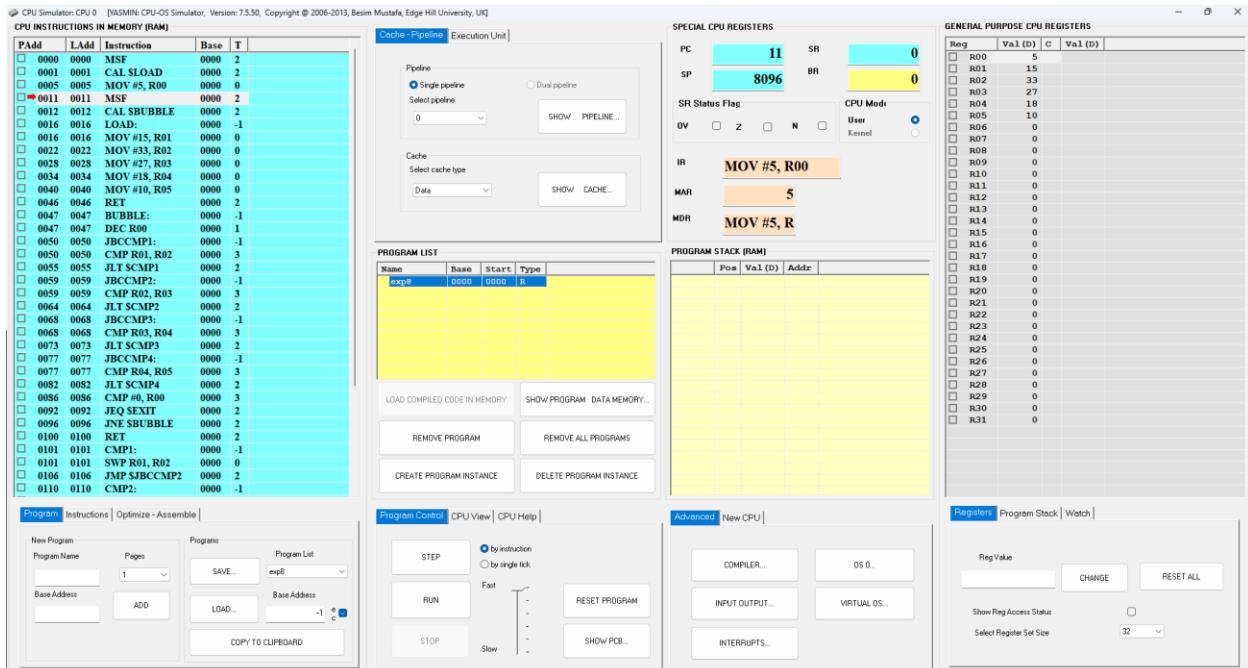


Fig: CPU Simulator Window (Before Sorting)

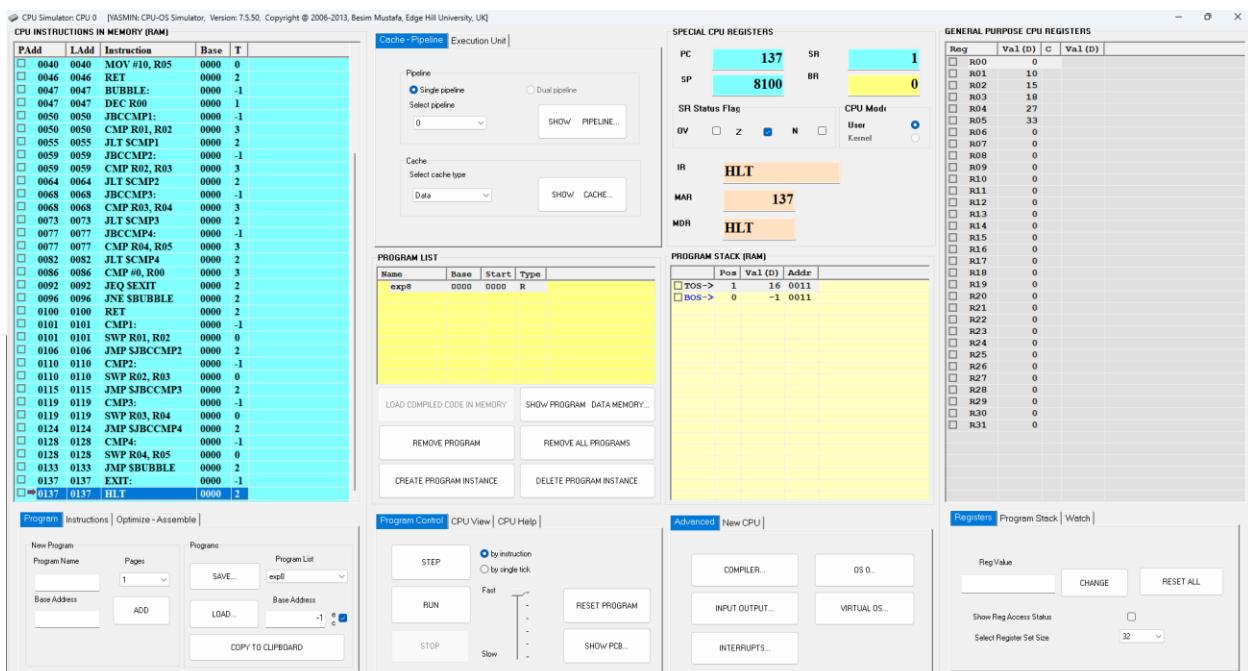


Fig: CPU Simulator Window (After Sorting)

Lab Sheet-Direct Mapped Cache

1. Objective:

At the end of this lab session, the student should be able to:

- Understand direct mapped cache
- Understand the behaviour of direct-mapped cache by varying block size and cache size

2. Theory:

Cache memory is a small amount of fast memory placed between the CPU and the main memory to bridge the gap in access times. Due to the locality of memory references, the cache memory can enhance the computer system's performance. The efficiency gained by using a cache memory varies depending on cache size, block size, and other cache parameters, but it also depends on the program and data.

The mapping function is the way of associating memory blocks with the cache line. Three mapping functions available are

- Direct mapped
- Associative
- Set associative

The direct mapping technique is the simplest way of associating main memory blocks with the cache lines. In this technique, block k of the main memory maps into block k modulo m of the cache, where m is the total number of blocks. Since more than one main memory block is mapped onto a given cache block position, contention may arise for that position. This situation may occur even when the cache is not full. Conflict is resolved by allowing the new block to overwrite the current resident block. So the replacement algorithm is trivial.

3. Steps to perform :

- . The following program should be used for the analysis.

LDB 00, R00
LDB 01, R01

LDB 02, R02
LDB 03, R03
LDB 04, R04
LDB 05, R05
LDB 06, R06
LDB 07, R07
LDB 08, R08
LDB 09, R09
LDB 10, R10
LDB 11, R11
LDB 12, R12
LDB 13, R13
HLT

The procedure to enter the above code into the simulator is as follows:

- Create a program name and enter a base address in the **Base Address** text box
- Enter CPU instructions in the program
- Press the “Cache-Pipeline” tab and select the cache type as “data cache”, as shown in Figure 1. Press the “SHOW CACHE...” button. A new window will be opened, as shown in Figure 2.

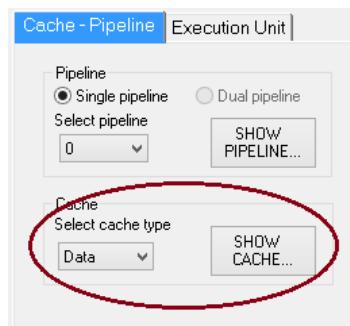


Figure 1: Cache – Pipeline setting

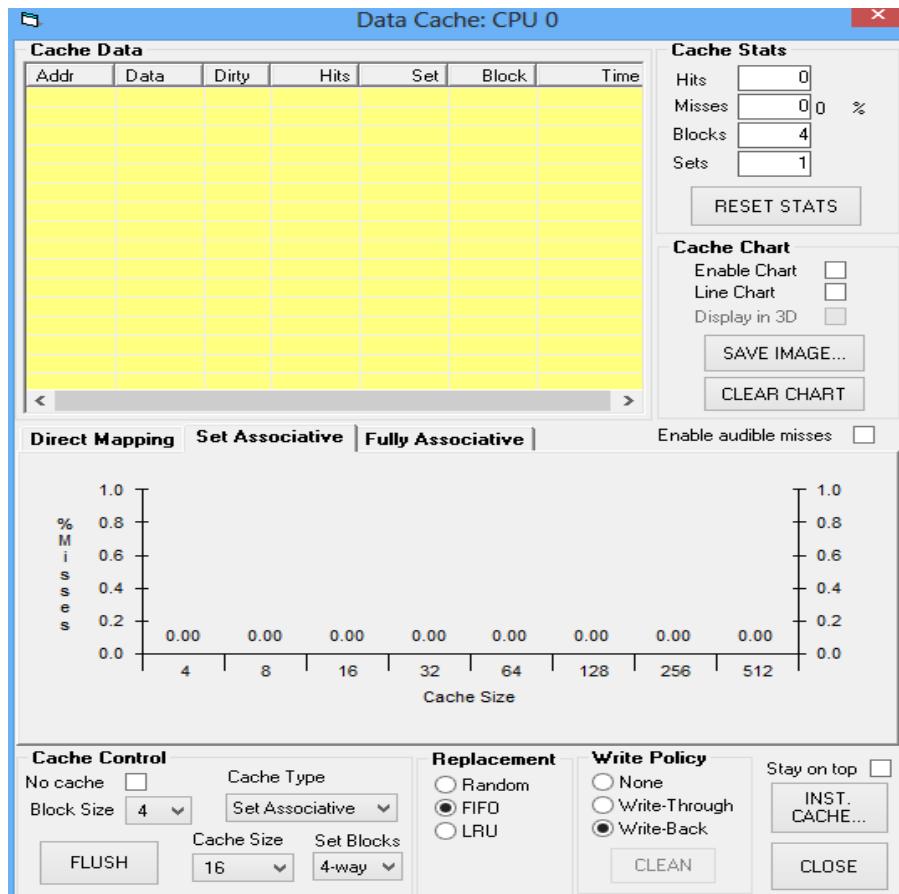


Figure 2: Data cache window

- (d) Set cache type to Direct mapped cache. Also, set block size, cache size and write policy.
- (e) Execute the program.
- (f) Note down cache miss, hit and compute the hit ratio.

Problem No 1:

Step 1: Set the following parameters in the cache control block of the data cache window, as shown in Figure 2.

- Block Size: 2
- Cache Size: 16 bytes
- Number of Blocks: 8 (automatically updated, which is equal to Cache size / Block Size)
- Cache Type: Direct Mapped
- Select Enable Chart check box
- Select stay on top check box

Step 2: Run the program in a single step and observe the changes on the data cache window

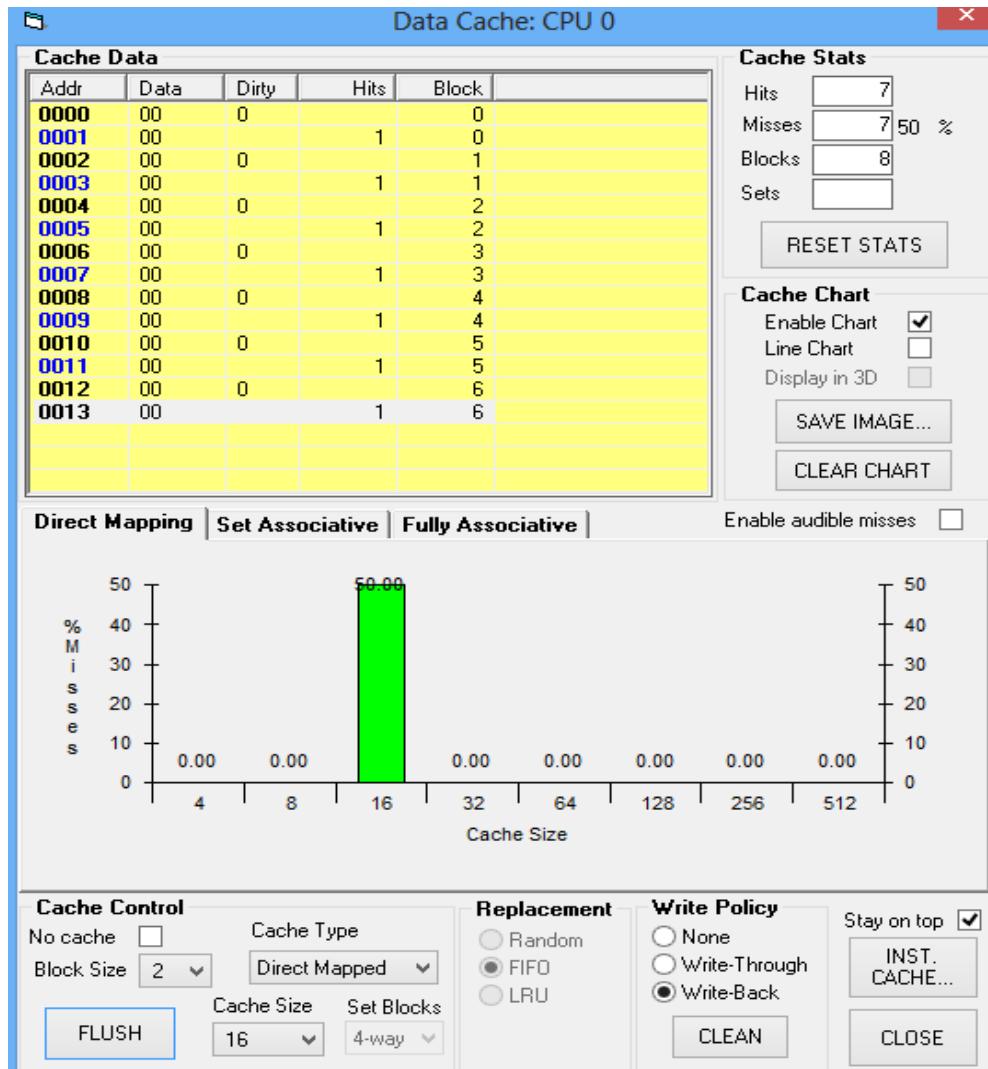


Figure 3: Data Cache window for problem no. 1

Step 3: Note down the number of hits, misses and hit ratio

Problem No 2: Analysis of direct-mapped cache by varying block size.

Set the following parameters:

- Blocks: 8
- Cache Type: Direct Mapped
- Cache Size: 16 bytes
- Select Enable Chart check box
- Select stay on top check box

Execute the above program by setting a block size to 2, 4, 8, 16 and 32. Record the observation in the following table.

Block Size	Number of blocks	Miss	Hit	Miss ratio
2	16	7	7	.5
4	16	4	10	0.285

8	16	2	12	0.142
16	16	1	13	0.0714
32	16	1	13	0.0714

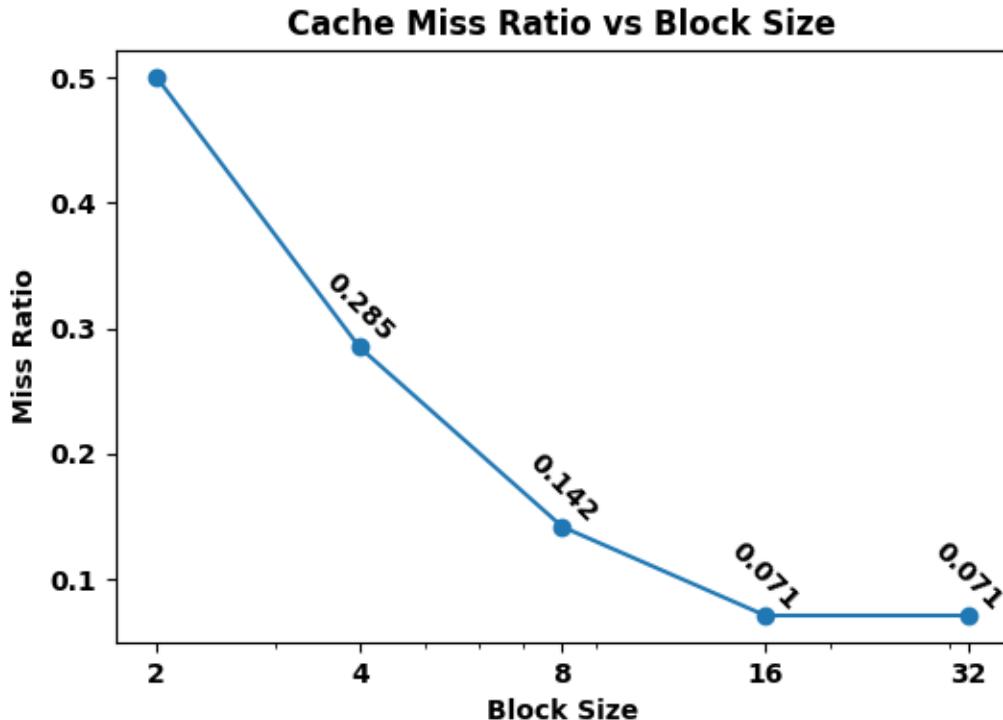


Figure 4: Graph of Cache miss ratio Vs Block size for problem no. 2

Problem No 3: Analysis of direct-mapped cache by varying cache size.

Set the following parameters:

- Block size: 2
- Cache Type: Direct Mapped
- Select Enable Chart check box
- Select stay on top check box

Execute the above program by setting a block size to 2, 4, 8, 16 and 32. Record the observation in the following table.

Block Size	Cache size	Miss	Hit	Miss ratio
2	4	7	7	0.5
2	8	7	7	0.5
2	16	7	7	0.5
2	32	7	7	0.5
2	64	7	7	0.5

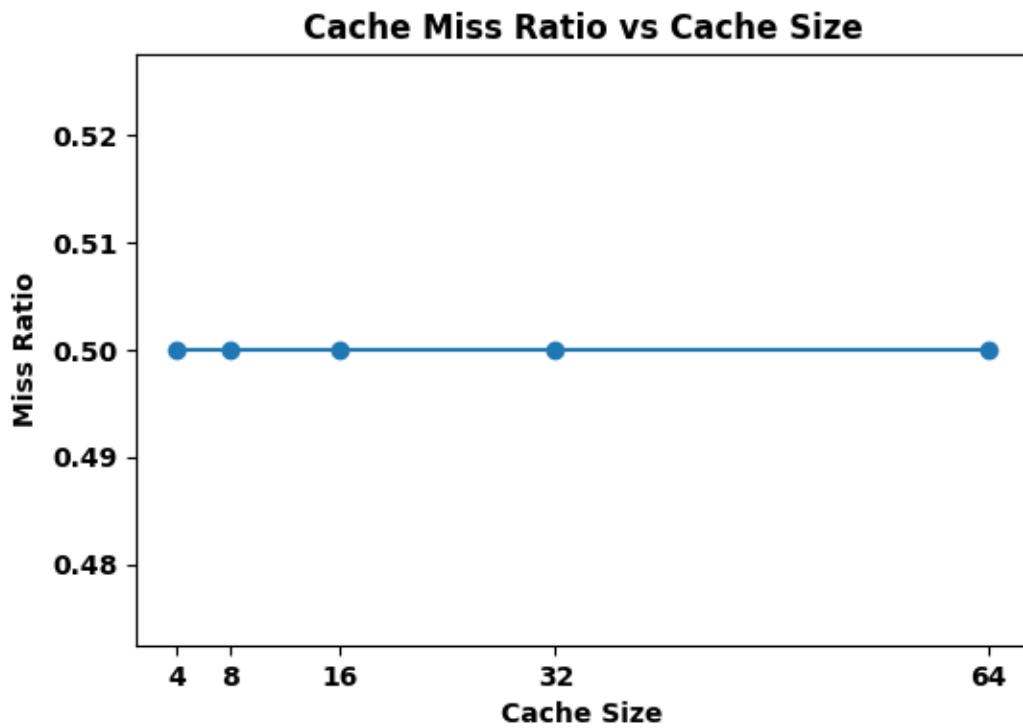


Figure 5: Graph of Cache miss ratio Vs Cache size for problem no. 3

Lab Sheet - Fully Associative Cache

Objectives

At the end of this lab session, the student should be able to:

- Understand fully associative mapped cache
- Understand the behaviour of fully associative mapped cache by varying block size and cache size
- Understand the behaviour of fully associative mapped cache concerning replacement algorithms

Prerequisites: Module No. 4 - Cache Memory Organization

6.0 Performance Analysis of Fully Associative Mapped Cache

The following program should be used to analyse a fully associative mapped cache.

LDB 00, R00

LDB 01, R01

LDB 02, R02

LDB 03, R03

LDB 04, R04

LDB 05, R05

LDB 06, R06

LDB 07, R07

LDB 08, R08

LDB 09, R09

LDB 10, R10

LDB 11, R11

LDB 12, R12

LDB 13, R13

HLT

Press the “Cache-Pipeline” tab and select the cache type as “data cache”, as shown in Figure 1. Press the “Show Cache...” button. A new window will be opened, which is shown in Figure 2.

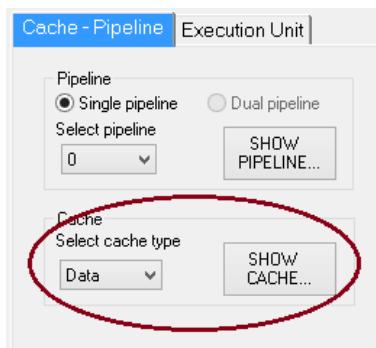


Figure 1: Cache – Pipeline setting

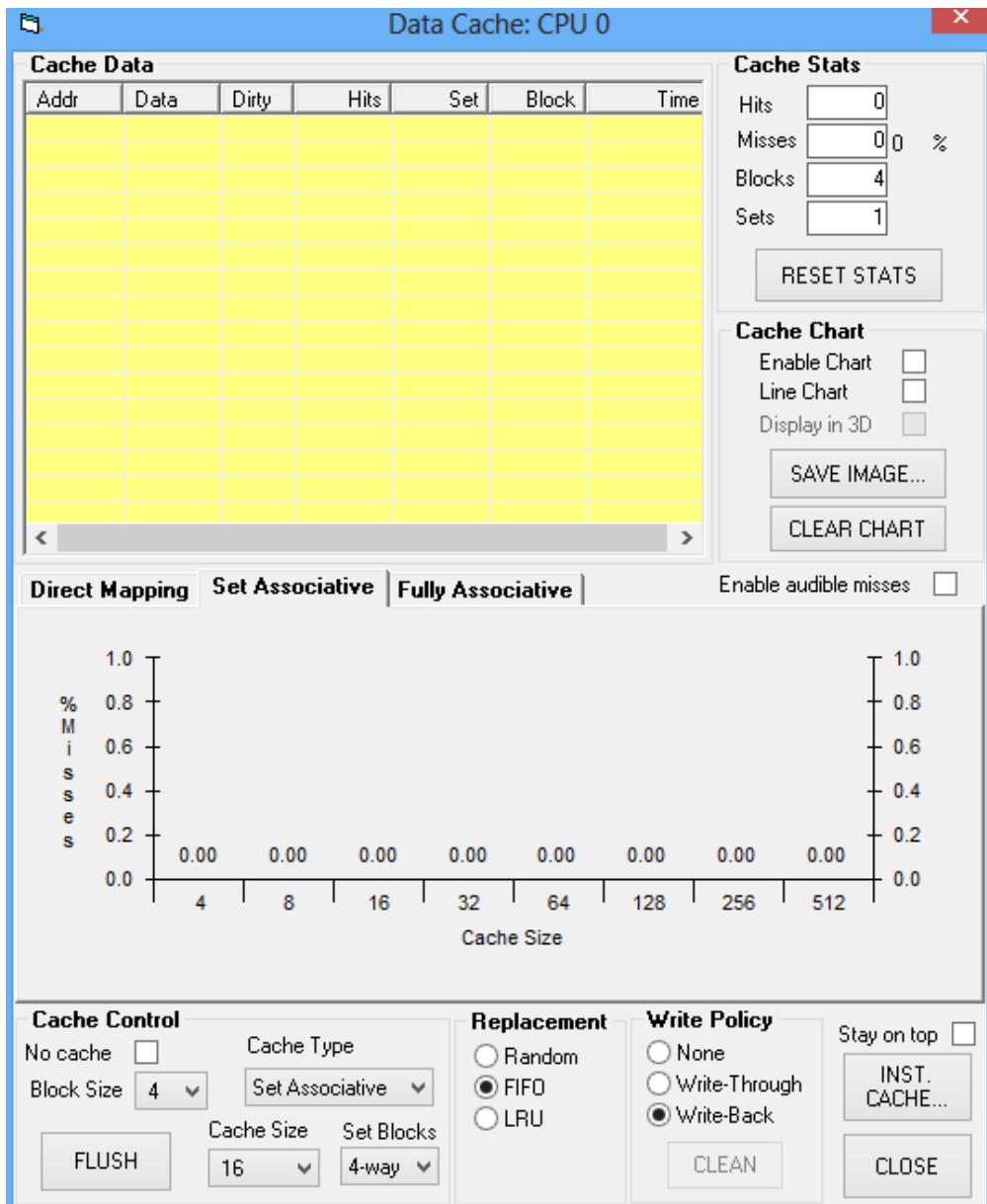


Figure 2: Data cache window

Problem 1 :

Step 1: Set the following parameters in the cache control block of the data cache window, as shown in Figure 2.

- Block Size: 2
- Cache Type: Fully Associative
- Cache Size: 16 bytes
- Select Enable Chart check box
- Select stay on top check box
- Replacement: Random

Step 2: Run the program in a single step and observe the changes on the data cache window

Step 3: Note down the number of hits, misses and hit ratio

Problem No 2: Analysis of Fully associative cache by varying block size.

Set the following parameters:

- Cache Type: Fully associative cache
- Cache Size: 16 bytes
- Select Enable Chart check box
- Select stay on top check box

Execute the above program by setting a block size to 2, 4, 8, 16 and 32. Record the observation in the following table.

Block Size	Cache Size	Miss	Hit	Miss ratio
2	16	7	7	0.5
4	16	4	10	0.285
8	16	2	12	0.142
16	16	1	13	0.0714
32	32	1	13	0.0714

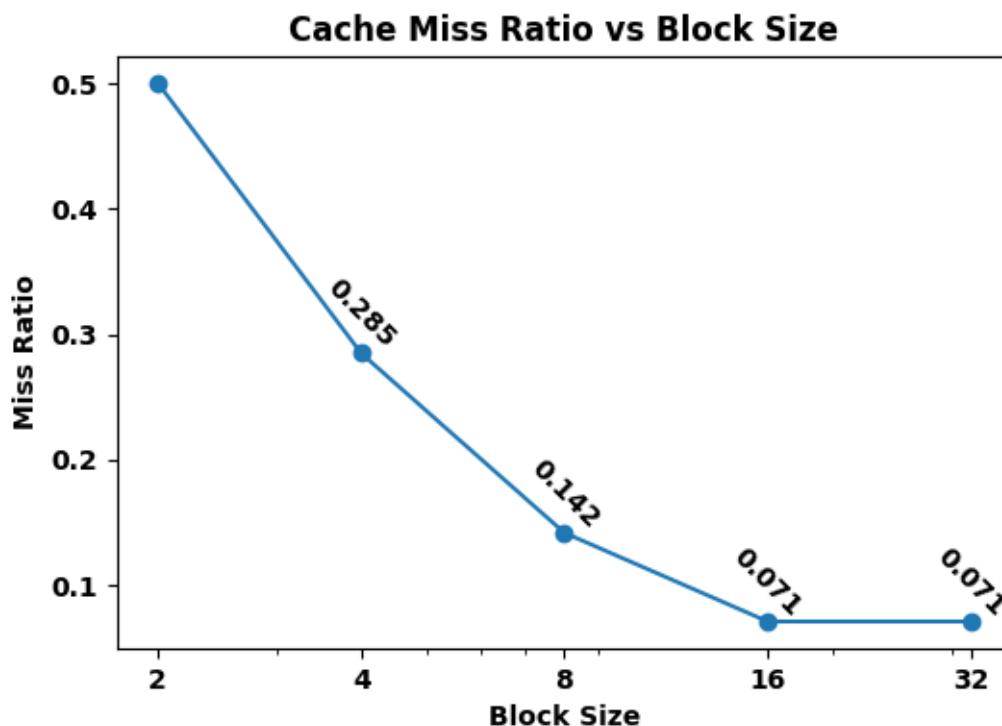


Figure 3: Graph of Cache miss ratio Vs Block size for problem no. 2

Problem No 3: Analysis of fully associative cache by varying cache size.

Set the following parameters:

- Block size: 2
- Cache Type: Fully associative cache
- Select Enable Chart check box
- Select stay on top check box
- Replacement: Random

Execute the above program by setting a block size to 2, 4, 8, 16 and 32. Record the observation in the following table.

Block Size	Cache size	Miss	Hit	Miss ratio
2	4	14	14	0.5
2	8	7	7	0.5
2	16	7	7	0.5
2	32	7	7	0.5
2	64	7	7	0.5

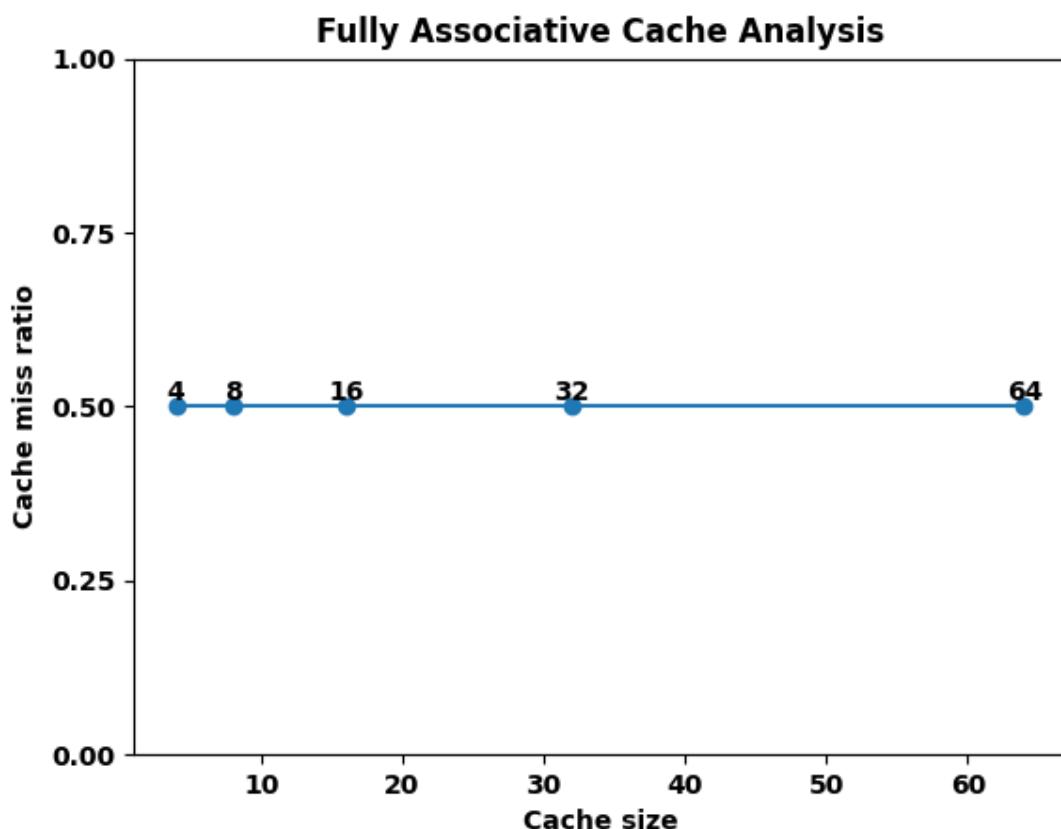


Figure 4: Graph of Cache miss ratio Vs Cache size for problem no. 3

Problem No 4: Analysis of fully associative cache concerning replacement algorithm.

Set the following parameters:

- Block size: 4
- Cache size: 8
- Cache Type: Fully associative cache
- Select Enable Chart check box
- Select stay on top check box

Execute the above program by setting the replacement algorithm as FIFO, Random and LRU. Record the observation in the following table.

Replacement Algorithm	Miss	Hit	Miss ratio
FIFO	4	10	0.285
LRU	4	10	0.285
Random	4	10	0.285

Are the results obtained identical for all the cases? Justify your answer.

Yes, the results obtained are identical for all the cases. The miss, hit, and miss ratio values are the same for all replacement algorithms used in the program. This suggests that for the given block size and cache size, the replacement algorithm does not have a significant impact on cache performance. However, it is important to note that this may not always be the case, and different replacement algorithms may have varying impacts on cache performance depending on the access patterns of the memory being used.

Problem No 5: Analysis of fully associative cache concerning replacement algorithm.

Set the following parameters:

- Block size: 4
- Cache size: 8
- Cache Type: Fully associative cache
- Select Enable Chart check box
- Select stay on top check box
- Save integer numbers from 0 to 40 in the main memory 0000 to 0040.

Execute the following program by setting the replacement algorithm as FIFO, Random and LRU. Record the observation in the following table.

LDB 01, R00
LDB 16, R01
LDB 32, R02
LDB 00, R03
LDB 16, R04
LDB 16, R05
LDB 32, R06
LDB 08, R07
LDB 04, R08
HLT

Replacement Algorithm	Miss	Hit	Miss ratio
Random	28	8	0.777
FIFO	8	1	0.888
LRU	8	1	0.888

Are the results obtained identical for all the cases? Justify your answer.

No, the results obtained are not identical for all the cases. The miss ratio obtained for each replacement algorithm is different. For example, the miss ratio obtained for the Random replacement algorithm is lesser compared to the other two algorithms. This is because the Random replacement algorithm selects a random block to replace, which may not be the least recently used or the oldest block. On the other hand, the FIFO and LRU replacement algorithms replace the least recently used block and the oldest block, respectively, which can lead to a lower hit rate and higher miss ratio. Therefore, the choice of replacement algorithm can have a significant impact on the performance of the cache system.

Lab Sheet - Set Associative Cache

Objectives

At the end of this lab session student should be able to:

- Understand set associative mapped cache
- Understand the behavior of set associative mapped cache by varying block size, cache size and number of sets
- Understand the behavior of fully associative mapped cache with reference to replacement algorithms

Prerequisites: Module No. 4 - Cache Memory Organization

Contents:

Performance analysis of Set Associative Mapped Cache

- 7.1 Effect of cache size on the performance of Set Associative Mapped Cache
- 7.2 Effect of block size on the performance of Set Associative Mapped Cache
- 7.3 Effect of number of sets on the performance of Set Associative Mapped Cache
- 7.4 Performance analysis of Set Associative Mapped Cache with various replacement algorithms

7.0 Performance analysis of Set Associative Mapped Cache

The following program should be used for the analysis of set associative mapped cache.

```
LDB 00, R00  
LDB 16, R01  
LDB 32, R02  
LDB 00, R03  
LDB 16, R04  
LDB 20, R05  
LDB 24, R06  
LDB 00, R07  
LDB 20, R08  
LDB 28, R09  
HLT
```

Press “Cache-Pipeline” tab and select cache type as “data cache” as shown in Figure 1. Press “Show Cache..” button. A new window will be opened which is as shown in Figure 2.

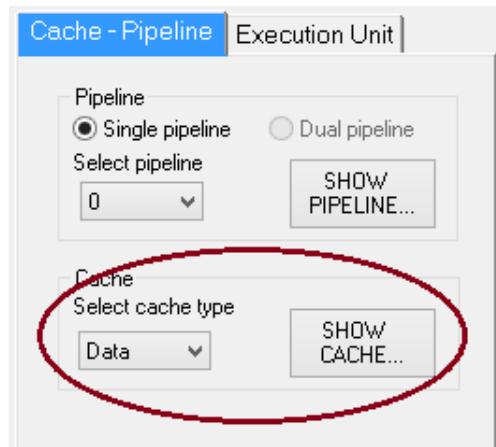


Figure 1: Cache – Pipeline setting

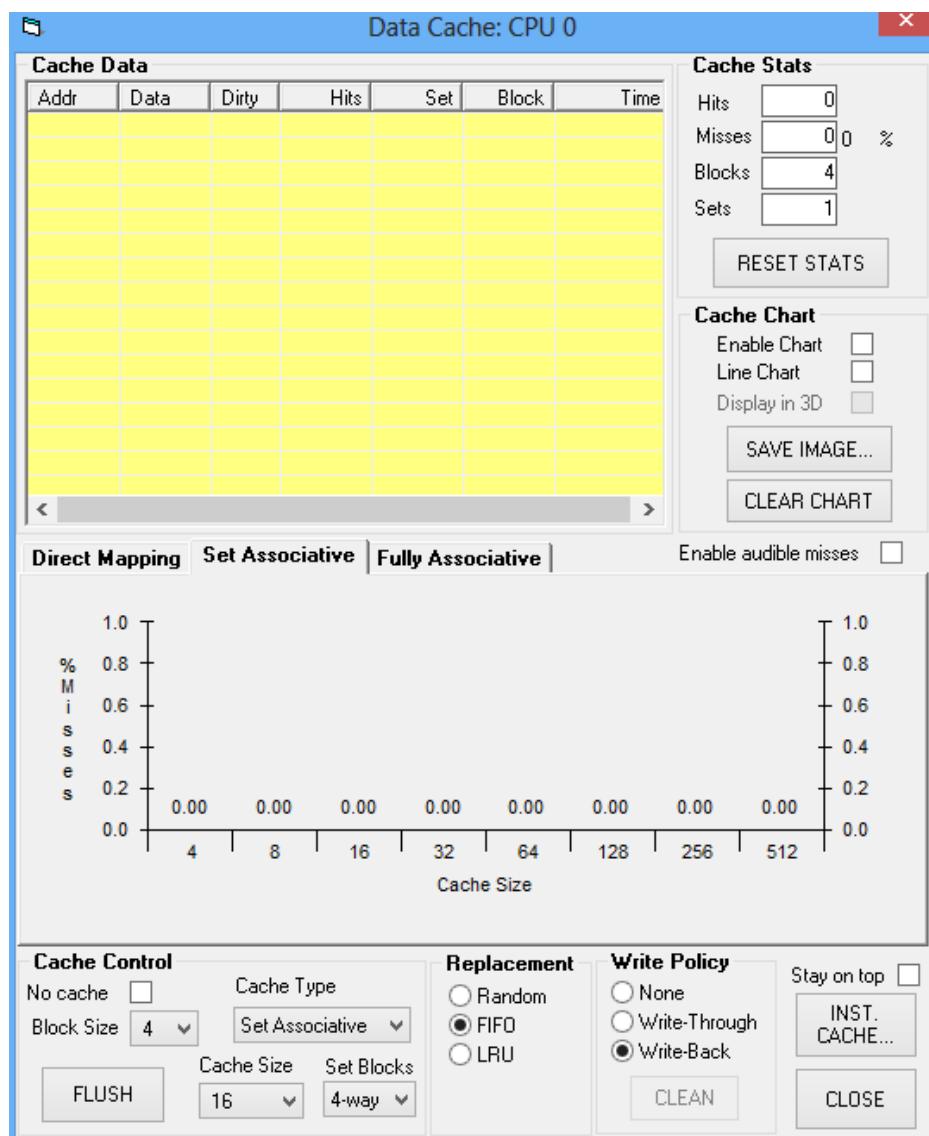


Figure 2: Data cache window

7.1 Effect of cache size on the performance of Set Associative Mapped Cache

Step 1: Set the following parameters available in cache control block of data cache window as shown in Figure 2.

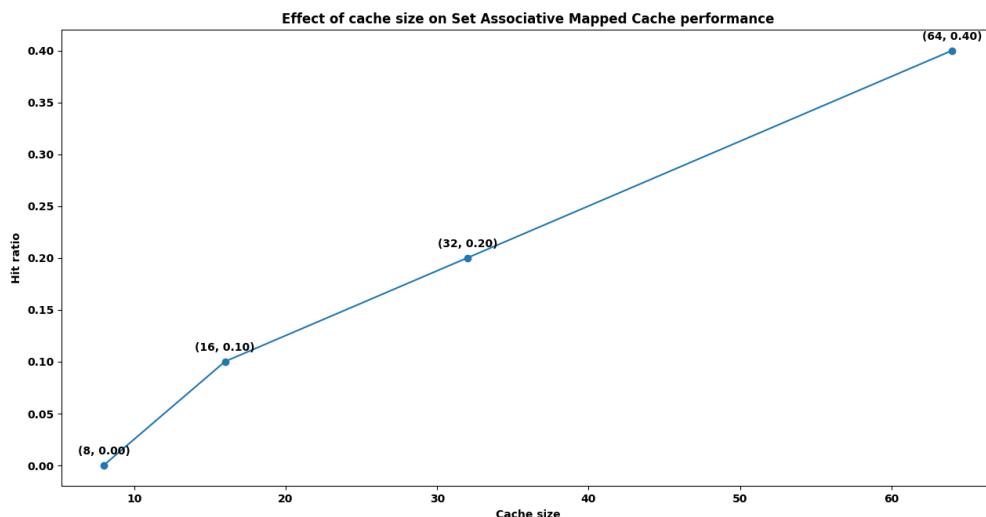
- Cache Type : Set Associative
- Number of sets (Set Blocks) : 2 way
- Block Size : 4
- Select Enable Chart check box
- Select stay on top check box
- Replacement: LRU

Step 2: Run the program in single step and observe the changes on the data cache window

Step 3: Fill in the following table with number of hits, misses and hit ratio with different cache sizes (8, 16, 32, and 64)

	Cache size = 8	Cache size = 16	Cache size = 32	Cache size = 64
Number of hit	0	1	2	4
Number of miss	10	9	8	6
Hit ratio	0	0.1	0.2	0.4

Step 4 : Plot the graph of cache size vs hit ratio and comment on the results obtained.



Based on the table and graph of cache size vs hit ratio, it is clear that increasing the cache size leads to a significant improvement in cache performance. As the cache size increases from 8 to 64, the number of hits increases from 0 to 4, and the hit ratio increases from 0 to 0.4. This suggests that larger caches are more effective at reducing cache misses and improving the overall performance of the Set Associative Mapped Cache.

However, it is worth noting that the improvement in hit ratio seems to level off beyond a cache size of 32, suggesting that there may be diminishing returns to further increasing the cache size beyond this point. Additionally, while the experiment provides valuable insights

into the effect of cache size on cache performance, it is important to keep in mind that the results may be specific to the particular parameters and conditions used in the experiment, and may not necessarily generalize to other settings.

Overall, the results obtained from the experiment suggest that cache size is an important factor to consider in optimizing the performance of a Set Associative Mapped Cache, and that larger cache sizes can significantly improve cache performance.

7.2 Effect of block size on the performance of Set Associative Mapped Cache

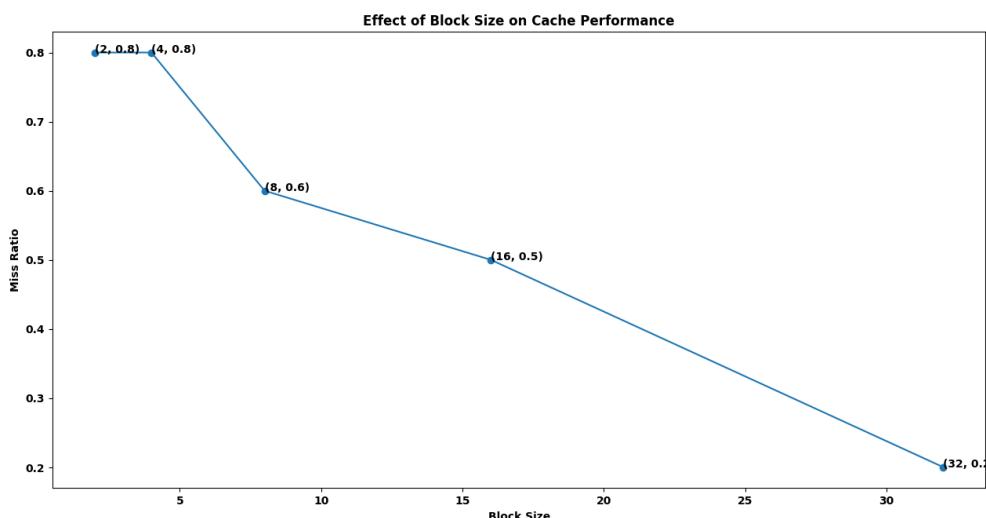
Step 1: Set the following parameters:

- Cache Type : Set Associative
- Cache Size : 32 bytes
- Set blocks : 2 way
- Select Enable Chart check box
- Select stay on top check box
- Replacement algorithm : LRU

Step 2: Execute the above program by setting block size to 2, 4, 8, 16 and 32. Record the observation in the following table.

Block Size	Miss	Hit	Miss ratio
2	8	2	0.8
4	8	2	0.8
8	6	4	0.6
16	5	5	0.5
32 (Cache Size : 64)	2	8	0.2

Step 3: Plot the graph of Cache miss ratio Vs Block size



7.3 Effect of number of sets on the performance of Set Associative Mapped Cache

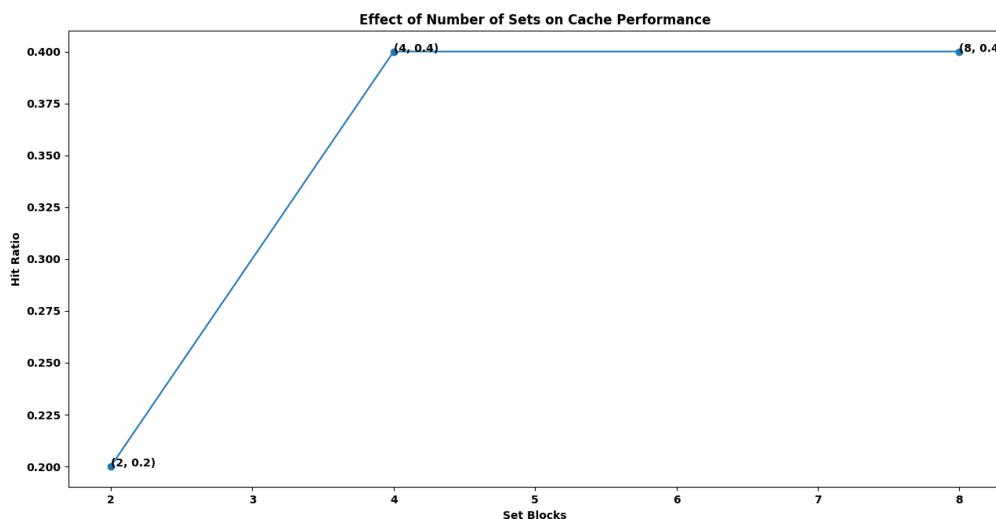
Step 1: Set the following parameters:

- Block size : 4
- Cache size : 32
- Cache Type : Set associative cache
- Select Enable Chart check box
- Select stay on top check box
- Replacement: LRU

Step 2: Execute the above program by setting set blocks to 2 way, 4 way and 8way. Record the observation in the following table.

Set Blocks	Miss	Hit	Hit ratio
2	8	2	0.2
4	6	4	0.4
8	6	4	0.4

Step 3: Plot the graph of Cache hit ratio Vs Set block size.



7.4 Performance analysis of Set Associative Mapped Cache with various replacement algorithms

Step 1 Set the following parameters:

- Block size : 4
- Cache size : 32
- Set blocks : 2 way
- Select Enable Chart check box
- Select stay on top check box

Step 2: Execute the above program by setting replacement algorithm as FIFO, Random and LRU. Record the observation in the following table.

Date:

Name: Ashvath S.P

Reg No: 2162014

Replacement Algorithm	Miss	Hit	Miss ratio
FIFO	8	2	0.8
LRU	8	2	0.8
Random	63	17	0.787