Assignment no.A-1

Roll no:-

- **Title**
  To implement calculator using concurrent Lisp.

- **Problem Definition**
  Implement an calculator (64 bit Binary Multiplication) application using concurrent lisp.

- **Learning Objectives**
  To understand Lisp programming
  To learn and implement concurrency concept using lisp.

- **Learning Outcome**
  Successfully implemented calculator by using concurrent Lisp.

- **Software and Hardware Requirement**
  Software Requirements

  - SBCL
  - Ubuntu 14.04 OS

  Hardware Requirements

  - Dual core CPU
  - 64 bit Processor

- **Theory:**

1. **Introduction**

   We are going to implement a 64 bit calculator using concurrent lisp. Here concurrent lisp is used in order to provide concurrency. Lisp is a programming language that stands for list processing the program itself is made up of lists. It uses fully parenthesized prefix notation. It is based on symbolic expressions or S-expressions. Calculator is implemented along with SBCL.

   SBCL is low level threading interface that maps onto host operating system's concept of threads or lightweight processes. It means that the thread may take the advantage of hardware multiprocessing on machines that have more than one CPU. This is found in SB-THREAD package. So, by using thread library we make threads and assign those threads to each operation in the program. Each thread processes simultaneously on data and output is produced.

2. **Related concept for Assignment LISP**

   - **LISP overview**

     John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implement by Steve Russell on an IBM 704 computer.
     It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively.

Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups, which were successors to Maclisp like ZetaLisp and NIL (New Implementation of Lisp) etc.
It serves as a common language, which can be easily extended for specific implementation. Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

- **Features of Common LISP**
  - It is machine-independent.
  - It uses iterative design methodology, and easy extensibility.
  - It allows updating the programs dynamically.
  - It provides high level debugging.
  - It provides advanced object-oriented programming.
  - It provides convenient macro system.
  - It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
  - It is expression-based.
  - It provides an object-oriented condition system.
  - It provides complete I/O library.
  - It provides extensive control structures.
- **Applications Built in LISP**
  - Emacs
  - G2
  - AutoCad
  - Igor Engraver
  - Yahoo Store
- **LISP functions**
  This section describes a number of simple operations on lists, i.e., chains of cons cells.
  - **cl-caddr x**
    This function is equivalent to (car (cdr (cdrx))). Likewise, this package defines all 24 cxxxr functions where xxx is up to four 'a's and/or 'd's. All of these functions are setf-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.
  - **cl-first x**
    This function is a synonym for (car x). Likewise, the functions cl-second, cl-third, ..., through cl-tenth return the given element of the list x.
  - **cl-rest x**
    This function is a synonym for (cdrx).
  - **cl-endp x**
    Common Lisp defines this function to act like null, but signaling an error if x is neither a nil nor a cons cell. This package simply defines cl-endp as a synonym for null.
  - **cl-list-length x**
    This function returns the length of list x, exactly like (length x), except that if x is a circular list (where the cdr-chain forms a loop rather than terminating with nil), this function returns nil. (The regular length function would get stuck if given a circular list. See also the safe-length function.)

- **cl-list * arg and rest others**
  This function constructs a list of its arguments. The final argument becomes the cdr of the last cell constructed. Thus, (cl-list* abc) is equivalent to (cons a (cons bc)), and (cl-list* ab nil) is equivalent to (list ab).

- **cl-ldiff list sublist**
  If sublist is a sublist of list, i.e., is eq to one of the cons cells of list, then this function returns a copy of the part of list up to but not including sublist. For example, (cl-ldiff x (cddr x)) returns the first two elements of the list x. The result is a copy; the original list is not modified. If sublist is not a sublist of list, a copy of the entire list is returned.

- **cl-copy-list list**
  This function returns a copy of the list list. It copies dotted lists like (1 2 . 3) correctly.

- **cl-tree-equal x y and key :test :test-not :key**
  This function compares two trees of cons cells. If x and y are both cons cells, their cars and cdrs are compared recursively. If neither x nor y is a cons cell, they are compared by eql, or according to the specified test. The :key function, if specified, is applied to the elements of both trees.

- **Lisp- Basic Syntax** LISP programs are made up of 3 basic building blocks:
  - atom
    - It is a number or string of contiguous characters. It includes special characters and numbers.
  - list
    - A list is a sequence of atoms and/or other lists enclosed in parantheses.
  - string
    - A group of characters eclosed in quotes
- **LISP operators**
  An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. LISP allows numerous operations on data, supported by various functions, macros and other constructs.
  The operations allowed on data could be categorized as:

  - Arithmetic Operations
  - Comparison Operations
  - Logical Operations
  - Bitwise Operations

- **Arithmetic Operations**
  The following table shows all the arithmetic operators supported by LISP.

Operator	Description	Example
. +	Adds two operands	(+AB) will give 30
-	Subtracts second operand from the first	(- A B) will give -10
*	Multiplies both operands	(* A B) will give 200
/	Divides numerator by de-numerator	(/ B A) will give 2
mod,rem	Modulus Operator and remainder of after an integer division	(mod B A)will give 0
Incf	Increments operator increases integer value by the second argument specified	(incf A 3) will give 13
Decf	Decrements operator decreases integer value by the second argument specified	(decf A 4) will give 9

- **Comparison Operations**
  Following table shows all the relational operators supported by LISP that compares between numbers. However unlike relational operators in other languages, LISP comparison operators may take more than two operands and they work on numbers only.

Operator	Description	Example
=	Checks if the values of the operands are all equal or not, if yes then condition becomes true.	(= A B) is not true.
/=	Checks if the values of the operands are all different or not, if values are not equal then condition becomes true.	(/= A B) is true.
>	Checks if the values of the operands are monotonically decreasing.	(> A B) is not true.
<	Checks if the values of the operands are monotonically increasing.	(< A B) is true.
>=	Checks if the value of any left operand is greater than or equal to the value of next right operand, if yes then condition becomes true.	(>= A B) is not true.
<=	Checks if the value of any left operand is less than or equal to the value of its right operand, if yes then condition becomes true.	(<= A B) is true.
Max	It compares two or more arguments and returns the maximum value.	(max A B) returns 20
Min	It compares two or more arguments and returns the minimum value.	(min A B) returns 20

- **Logical Operations on Boolean Values**
  Common LISP provides three logical operators: and, or, and not that operates on Boolean values.

Operator	Description	Example
And	It takes any number of arguments. The arguments are evaluated left to right. If all arguments evaluate to non-nil, then the value of the last argument is returned. Otherwise nil is returned.	(and A B) will return NIL.
Or	It takes any number of arguments. The arguments are evaluated left to right until one evaluates to non-nil, in such case the argument value is returned, otherwise it returns **nil**.	(or A B) will return 5.
Not	It takes one argument and returns **t** if the argument evaluates to **nil**.	(not A) will return T.

- **Lisp Macros**
  - Macros allow you to extend the syntax of standard LISP
  - A macro is a function that takes an s-expression as arguments and returns a LISP form, which is then evaluated.
  - In LISP, a named macro is defined using another macro named defmacro.
  - (defmacro macro-name (parameter-list))
    "Optional documentation string."
    body-form
- **Lisp Functions**
  - The macro named defun is used for defining functions. The defun macro needs three arguments:
  - 1.Name of function.
  - 2.Parameter of function.
  - 3.Body of function.
  - (defun funct-name (parameter-list)
    "Optional documentation string."
    body-form)
- **Lisp Arrays**
  - LISP allows you to define single or multiple-dimension arrays using the make-array function. Example, to create an array with 10- cells, named my-array
    (setf my-array (make-array '(10)))
  - To access the content of the tenth cell
    (aref my-array 9)
- **Lisp Strings**
  - Strings in Common Lisp are vectors, i.e., one-dimensional array of characters.
  - String literals are enclosed in double quotes
  - Any character supported by the character set can be enclosed within double quotes to make a string, except the double quote character and the escape character.
  - However, you can include these by escaping them with a backslash.
- **SBCL**

  Steel Bank Common Lisp (SBCL) is a high performance Common Lisp compiler. It is open source / free software, with a permissive license. In addition to the compiler and runtime system for ANSI Common Lisp, it provides an interactive environment including a debugger, a statistical profiler, a code coverage tool, and many other extensions.

SBCL runs on a number of POSIX platforms, and experimentally on Windows. See the download page for supported platforms, and getting started guide for additional help.

– **Starting sbcl**
To run SBCL type sbcl at the command line.

You should end up in the toplevel REPL (read, eval, print -loop), where you can interact with SBCL by typing expressions.

sbcl This is SBCL 0.8.13.60, an implementation of ANSI Common Lisp.
More information about SBCL is available at ¡http://www.sbcl.org/¿.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under BSD-style licenses.
See the CREDITS and COPYING files in the distribution for more information.

– **Stopping sbcl**

SBCL can be stopped at any time by calling sb-ext:exit, optionally returning a specified numeric value to the calling process. See Threading for information about terminating individual threads.

Function: exit [sb-ext] key code abort timeout

Terminates the process, causing sbcl to exit with code. code defaults to 0 when abort is false, and 1 when it is true.

– **Threading basics**

(make-thread (lambda () (write-line "Hello, world")))

* **Structure: sb-thread:thread**

Class precedence list: thread, structure-object, t
Thread type. Do not rely on threads being structs as it may change in future versions.
* **Variable: sb-thread:*current-thread***

Bound in each thread to the thread itself.
* **Function: sb-thread:make-thread function key name**

Create a new thread of name that runs function. When the function returns the thread exits.
* **Function: sb-thread:thread-alive-p thread**

Check if thread is running.
* **Function: sb-thread:list-all-threads**

Return a list of the live threads.
* **Condition: sb-thread:interrupt-thread-error**

Class precedence list: interrupt-thread-error, error, serious-condition, condition, t Interrupting thread failed.
* **Function: sb-thread:interrupt-thread-error-thread condition**

The thread that was not interrupted.

* **Function: sb-thread:interrupt-thread thread function**

  Interrupt the live thread and make it run function. A moderate degree of care is expected for use of interrupt-thread, due to its nature: if you interrupt a thread that was holding important locks then do something that turns out to need those locks, you probably won't like the effect.
* **Function: sb-thread:terminate-thread thread**

  Terminate the thread identified by thread, by causing it to run sb-ext:quit - the usual cleanup forms will be evaluated

– **Threading Objects**

* **Structure: thread [sb-thread]**

  Class precedence list: thread, structure-object, t
  Thread type. Do not rely on threads being structs as it may change in future versions.
* **Variable: *current-thread* [sb-thread]**

  Bound in each thread to the thread itself.
* **Function: list-all-threads [sb-thread]**

  Return a list of the live threads. Note that the return value is potentially stale even before the function returns, as new threads may be created and old ones may exit at any time.
* **Function: thread-alive-p [sb-thread] thread**

  Return t if thread is still alive. Note that the return value is potentially stale even before the function returns, as the thread may exit at any time.
* **Function: thread-name [sb-thread] instance**

  Name of the thread. Can be assigned to using setf. Thread names can be arbitrary printable objects, and need not be unique.
* **Function: main-thread-p [sb-thread] optional thread**

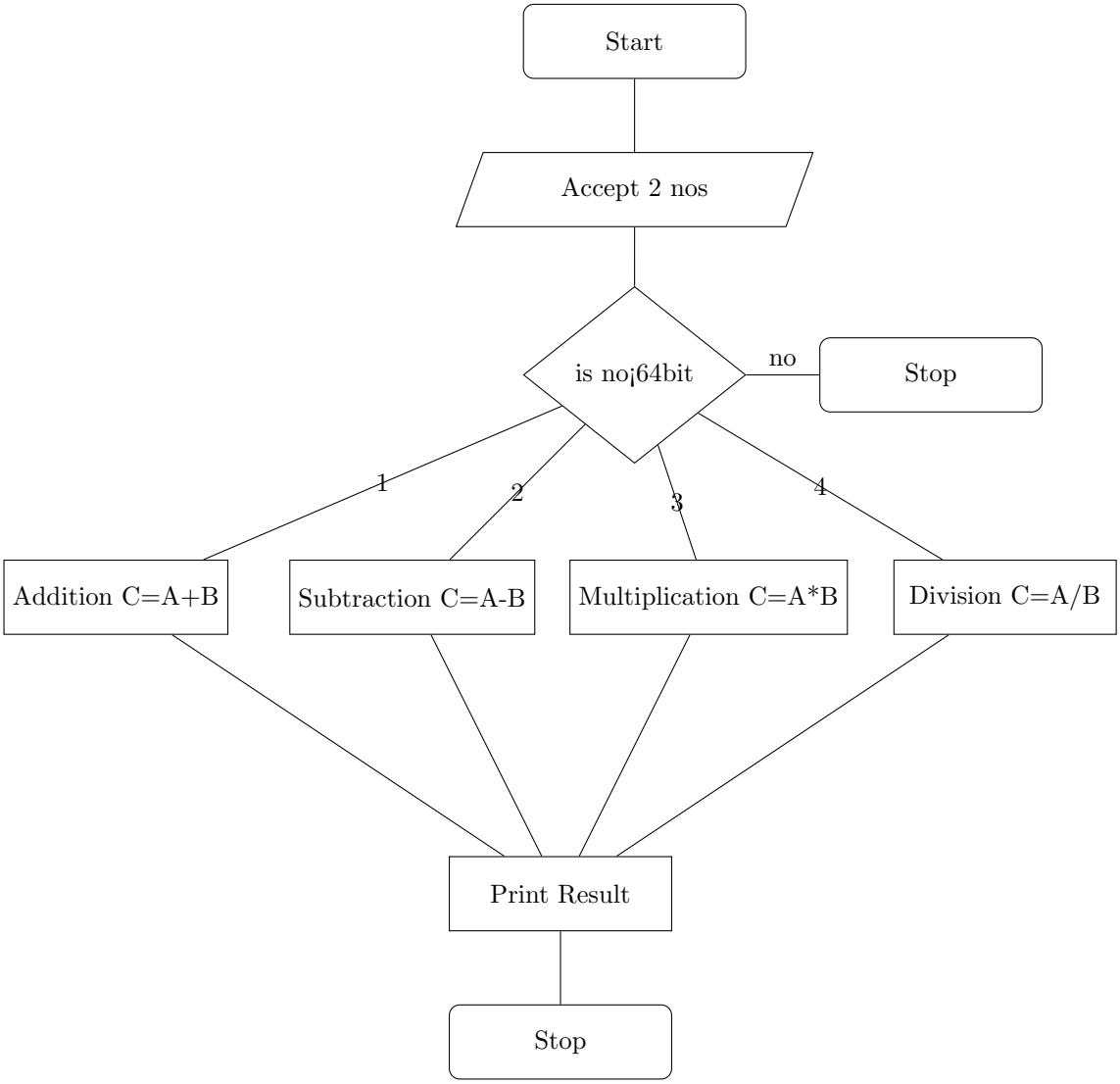
  True if thread, defaulting to current thread, is the main thread of the process.
* **Function: main-thread [sb-thread]**

  Returns the main thread of the process.

3. **Algorithm**

(a) Start
(b) Initialize three variables i.e. A, B, C
(c) Accept the values from user i.e. A and B
(d) Accept choice from user
(e) If choice == 1 Addition i.e C=A+B
(f) If choice == 2 Subtraction i.e C=A-B
(g) If choice == 3 Multiplication i.e C=A*B
(h) If choice == 4 Division i.e C=A/B
(i) Print the result
(j) Stop

4. **Flow Chart**



5. **Mathematical model:**

$$In \quad Concurrent\_Lisp \quad ,$$

$$Let \quad U \quad = \{D \quad , ND \quad , S \quad , F \quad , f \quad , In \quad , Out\}$$

$$where$$

$$U: \quad Universal \quad set$$

$$D: \quad Deterministic \quad data \quad values \quad i.e. \quad is \quad initial \quad values$$

$$ND: \quad Non \quad deterministic \quad data \quad values \quad i.e. \quad values \quad given \quad by \quad user.$$

$$S: \quad Cases \quad for \quad success$$

$$F: \quad Cases \quad for \quad failure$$

$$f: \quad calculation$$

$$In: \quad Input \quad data$$

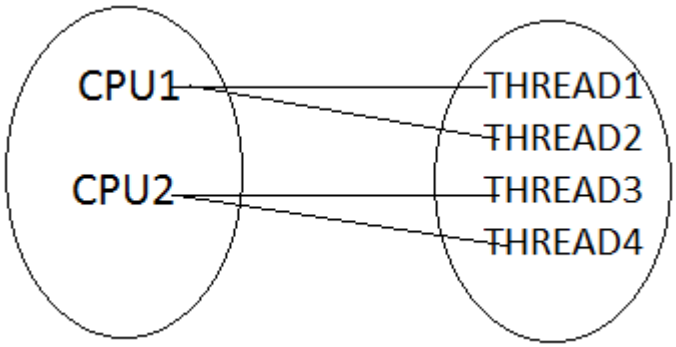
$$Out: \quad Output \quad data$$

$$In: \quad Integer\_1 \quad , Integer\_2$$

$$Out: \quad Addition, \quad Subtraction, \quad Multiplication, \quad Division$$

$$f:$$

Venn Diagram:



$$\begin{aligned} Addition\_of\_2\_nos &= (A + B) \\ Subtraction\_of\_2\_nos &= (A - B) \\ Multiplication\_of\_2\_nos &= (A * B) \\ Divison\_of\_2\_nos &= (A / B) \end{aligned}$$

$$S: \quad no \quad should \quad not \quad be \quad > 64 \quad and \quad not \quad < 0.$$

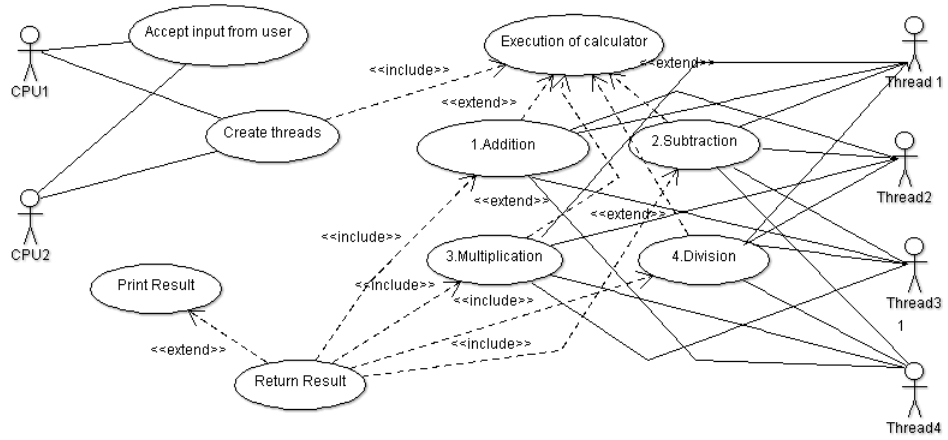
$$F: \quad If \quad success \quad conditions \quad are \quad not \quad satisfied.$$

6. **Software Requirement Specification**

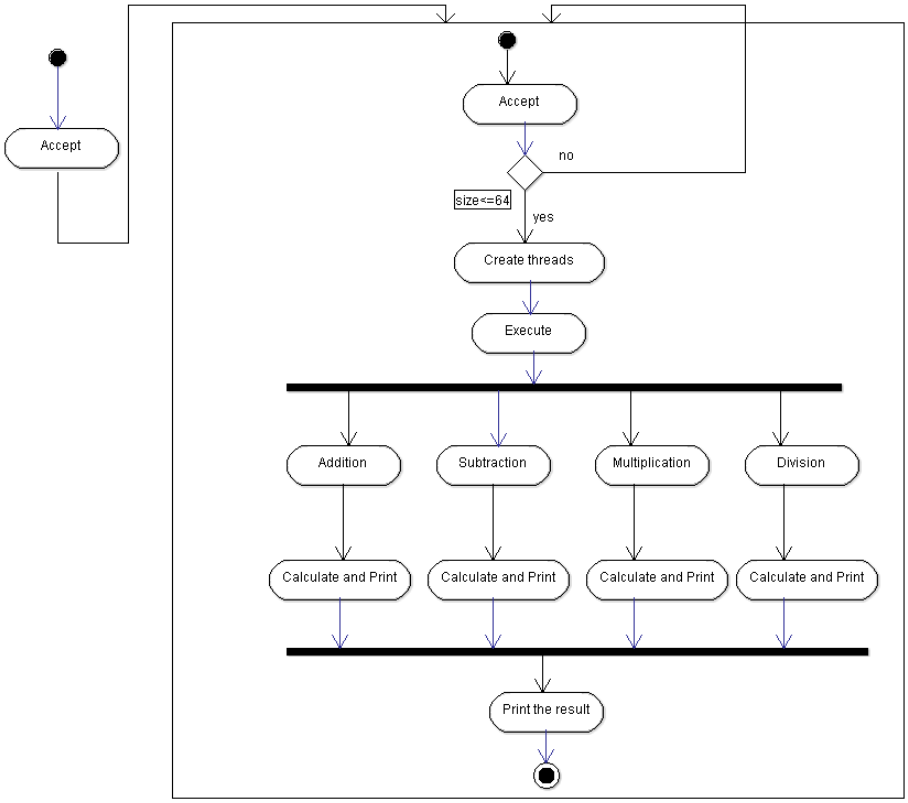
Software Specification	Description
1.Project Scope	To design a calculator which performs 64 bit binary operations.
2. Functional requirements	Input given must be valid. Operations are carried out on smoothly
3. Non functional requirements	If the no. exceeds then integer arithmetic operations. Quality, usability, performance
4. Design	The no. should not exceed than 64 binary or else fail.
5. Implementation	Using concurrent lisp and by using thread concept.
6.Hardware requirements	Dual core CPU, i.e. 4 threads.
7. External interfacing	No external interfacing required

7. UML Diagrams

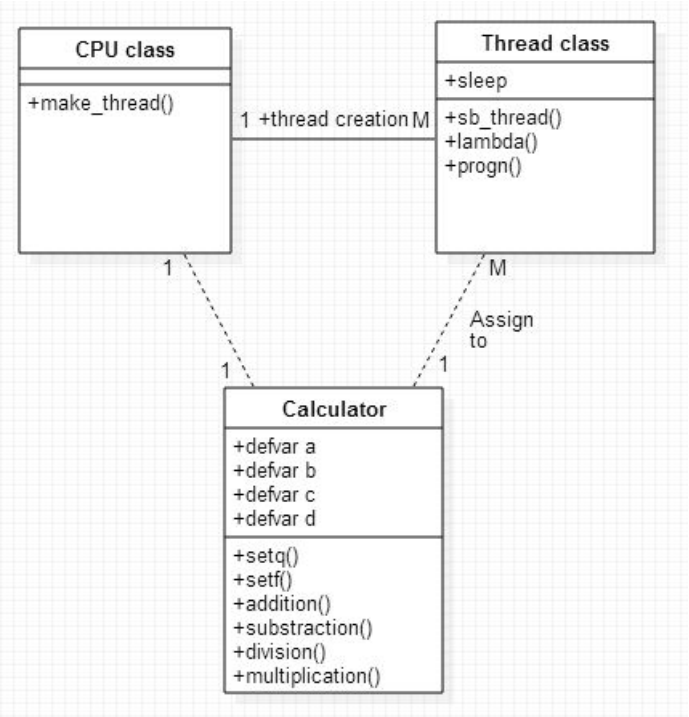
(a) Use Case Diagram



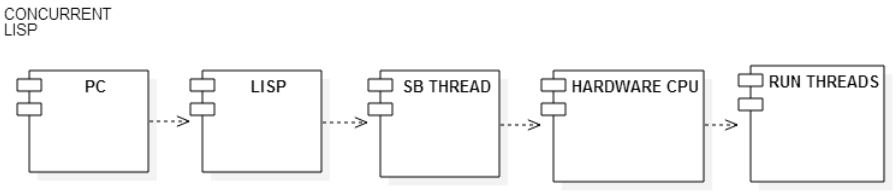
(b) Activity Diagram



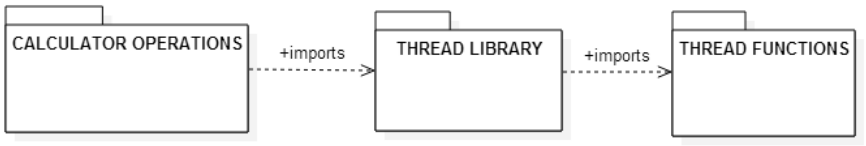
(c) Class Diagram



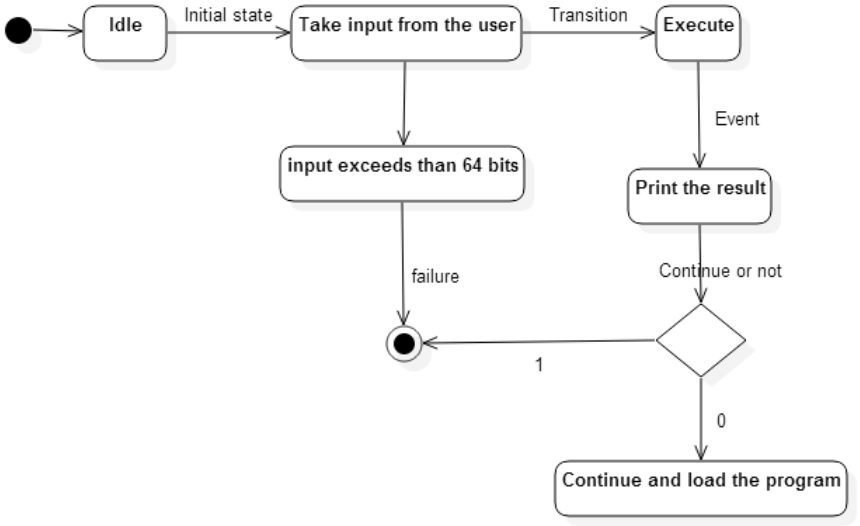
(d) Deployment Diagram



(e) Package Diagram



(f) State And Transition Diagram



- **Input**

  Inputs to this program are two integer numbers.

- **Output**

  Output of this program is binary number.

- **Conclusion**

  Hence in this way we have implemented a calculator (64 bit Binary Multiplication) application using concurrent lisp.

Course Outcome

Course Objective.	Achieved outcome (Tick √)
CO I: Ability to perform multi-core, Concurrent and Distributed Programming.	√
CO II: Ability to perform Embedded Operating Systems Programming using Beaglebone.	
CO III: Ability to write Software Engineering Document.	√
CO IV: Ability to perform concurrent programming using GPU.	

Oral Questions

1. What do you mean by Lisp?
2. What are Features of Lisp?
3. How concurrency can be achieved by using Lisp?
4. What are functions in Lisp?
5. What is SBCL?
6. Explain Different functions in SBCL?
7. What are advantages And Disadvantages of Lisp?
8. Write Applications Of Lisp?
9. What are Different Operations performed using Lisp?

   Questions on Program

10. How to convert Decimal to Binary ?
11. What is meant by Progn() Function?
12. What do you mean by Lambda() expression?
13. What Sleep method does?