C++/SystemC Assignments

Vayavya Labs Pvt. Ltd.

Version 1.5, Mar 22 2019 / 18:27:11

1. General Information

1.1. Copyright Notice

1.2. Dos and don’ts

You are required to address the assignments described in this document yourself. External help should NOT be solicited - either from an individual, a company/organization or online forums. It is our expectation that you are able to carry out the tasks autonomously.

Referring to a language reference manual is allowed and encouraged. You can also consult external forums for pointed queries (e.g. "How to set file buffering to NONE in Python?") - but you are expected to completely understand the code that you finally submit. Blind copy-paste is not expected and defeats the purpose of this exercise. Even the code-snippets in this document must be typed out again and not copy-pasted.

The assignment implementation should be cleanly separated into header/source files. Use `.cpp` extension for C++ sources, `.h` for C++ header implementation. Other normal guidelines apply: use meaningful names, indent code correctly, provide comments. Use only spaces for indentation.

Each assignment should be done along with one or more test-case. The tests must be done in a separate file - and must (generally) run with no manual intervention. Provide a makefile for compiling sources to an executable. Sources should compile with no warnings (using `-Wall` with `g++`).

If you have any feedback or questions, please direct them to your contact person in Vayavya Labs.

2. Assignments

2.1. A simplified `std::list`

References:

- `std::list` reference: https://en.cppreference.com/w/cpp/container/list
- PIMPL idiom: https://en.cppreference.com/w/cpp/language/pimpl

Implement a class `simple_list` with the following features (refer to `std::list` documentation for details on what the methods should do):

- Is a list of `int` types
- Supports forward iteration (uses a singly linked list underneath)
- Uses `new`/`delete` internally for allocation / deallocation
- Provides a default constructor, destructor, assignment operator
- Provides `empty` and `max_size` methods
- Provides the following iterator methods: `begin`, `end`
- Provides the `front`, `push_front`, `erase_after` and `insert_after` methods
- Provide the required overloads so that `std::cout << list_obj` will print the list contents to `stdout`

The class should be implemented in `simple_list.cpp`/`simple_list.h` files. The test case should in a separate file, `test_simple_list.cpp`.

Additional challenges:

- Modify `simple_list` to use the PIMPL idiom - by moving implementation specific aspects to an other class, `simple_list_impl`.
- Modify `simple_list` to be a template class - templatized on the element type. What happens when the body of the methods are left in the `simple_list.cpp` file?

2.2. 40-bit integer type

Implement a datatype called `acc40` that behaves as a 40 bit signed integer with following features:

- Support following arithmetic operators: binary `+`, binary `-`, unary `-`, `*`, `/`
- Support following logical operators: `&&`, `||`
- Support relational operators: `!=`, `==`, `<`, `>`, `<=`, `>=`

- Support assignment operator
- Have a default constructor (sets value to 0) and copy constructor
- Conversion from/to integer types must be supported
- Provide the required overloads so that `std::cout << acc40_obj` will print the value to `stdout`

The class should be implemented in `acc40.cpp`/`acc40.h` files. The test case should be in a separate file, `test_acc40.cpp`.

Additional challenges:

- How will you modify the class to support overflow detection? i.e., when result of an arithmetic operation is more than 40 bits, a flag must be set (can be obtained using, `is_overflow` method on the object).
- If the task were to create a general template class, `template class acc<int N>` for N-bit arithmetic, how will the `acc40` implementation change? Would you consider any template specializations for optimizations here? Consider cases like `acc<1>` vs `acc<128>`. Actual implementation not required - providing a design approach would be sufficient.

2.3. Complex number implementation

Implement a class called `cplx` that has the following features:

- Has two `double` member elements to store the real/imaginary or magnitude/angle formats. A third `char` member element will indicate whether the complex number is stored in rectangular or polar format - 'r' for rectangular and 'p' polar. If the complex number is instantiated using the default constructor, this member element should be set to 'u' (undefined).
- Default constructor, constructor that accepts two `double` args and one char (constant 'r', or constant 'p') and also the copy constructor.
- Overload the assignment operator
- Overload the basic arithmetic operators: `+`, `-`, `*` and `/`. Ensure the overloaded operators takes care of invalid inputs. Note that the complex numbers could be in different formats and should be taken care of by the overloaded operators.
- Overload the insertion and extraction operators to accept `cplx` objects both in the polar and rectangular operators. Ensure that the display should display rectangular and polar formats appropriately.
- Also overload the `==` and `!=` relational operators to check for the equality or inequality of two complex numbers. The numbers being compared could be in different formats. The class should be implemented in `cplx.cpp`/`cplx.h` files. The test case should be in a

separate file called `test_cplx.cpp`.

2.4. Expression Trees

References:

- Binary Expression Trees:
  - https://en.wikipedia.org/wiki/Binary_expression_tree
  - https://web.archive.org/web/20170119094603/http://www.brpreiss.com/books/opus5/html/page264.html
- Visitor Pattern:
  - https://cpppatterns.com/patterns/visitor.html
  - https://gieseanw.wordpress.com/2018/12/29/stop-reimplementing-the-virtual-table-and-start-using-double-dispatch/

Consider an arithmetic binary expression with:

- Literal numbers, e.g.: `100`, `-42`)
- Arithmetic operators: binary `+`, binary `-`, unary `-`, `*`, `/`
- Variable references (following C-syntax), e.g.: `foo`, `bar42`

Implement an expression tree to store a representation of such an arithmetic expression. Some code-snippets are provided below as a start - modify/extend them as needed for the exercise. Implement constructors (default, copy), destructor if required. Note that when a `Tree` object gets deleted, it should free all the sub-nodes as well.

```
// Base class for all nodes
class Node
{
   // ...
};

class Num : public Node
{
   public:
      LiteralNumber(int n);
};

class Var : public Node
{
   // ...
};

class OpPlus : public Node
{
   // ...
};

// etc...

class Tree
{
   public:
      Tree(Node * current, Tree *left = NULL, Tree *right = NULL);
      Tree * left();  // Return left branch, NULL if empty
      Tree * right(); // Return right branch, NULL if empty
                      // Note: right branch for unary operators is NULL
      Node * node();  // Return current node reference
   private:
      // ...
};
```

An expression such as `1+2*3` can be represented as a tree as follows:

```
Tree t(new OpPlus
      , new Tree(new Num(1))
      , new Tree(new OpMultiply
         , new Tree(new Num(2))
         , new Tree(new Num(3))));
```

Overload the `operator<<` on `ostream` so a `Tree` object can be printed. How will you design the classes to let each class derived from `Node` to implement the "print" for that node?

```
std::cout << t << std::endl; // Should print: (1+(2*3))
```

Add a constructor to `Tree` class that constructs the tree from an prefix notation:

```
// Declaration:
   Tree::Tree(std::vector<Node *> prefix_expr);

// Usage:
   std::vector<Node *> expr = {new OpPlus
      , new Num(1)
      , new OpMultiply
      , new Num(2)
      , new Num(3)};
   Tree t(expr);
```

Add an `evaluate` method to the `Tree` class that will simplify the expression by evaluating all sub-trees formed only by operations on literal numerics.

```
// Declaration:
   Tree::evaluate();

// Usage:
   // Before, t = a + 1 + 2 + b*(2*3) - a
   t.evaluate();
   // After, t = a + 3 + b*6 - a
   // Note that the "a" term is not cancelled out
   // Objective is to only evaluate sub-trees formed using literal numerics
```

2.5. Simple Filesystem

In Unix, we know the philosophy, "Everything is a file". So a wide range of I/O resources such as documents ("regular files" - like text files, images, etc), directories, hard-drives, keyboards, printers, and other peripherals, etc are all visible as files.

For the purpose of this assignment, we will consider only the following types of files:

- Regular files / documents
- Directories - which may have other files or other directories
- Soft links - that refer to an other file or a directory

All files have some common attributes:

- A name that, for the purpose of this assignment, is alphanumeric with dot (`.`) as a separator
- A parent directory that specifies the directory under which the file is present

In a real file system, files have many additional attributes: creation time, access time, modification time, file permissions, file size, etc. We ignore those here.

In this assignment, you need to develop few utility C++ classes for file management - that mimic the following Linux operations:

- `ls f` - list the specified file.
  - If `f` is a regular file, just print the file name
  - If `f` is a directory, print file name of the directory contents
  - If `f` is a link, print file name of the link and the filename it links to (along with path).
- `touch f` - create a regular file in a specified directory
- `mkdir d` - create a directory in a specified directory
- `ln -s f1 f2` - create a soft-link to an existing file
- `mv f1 f2` move a file
- `rm f` - delete a file or a soft-link
- `rm -r d` - delete directory recursively

There is a special directory, `root`, that doesn't have user-specified name and is its own parent. The `root` directory can't be created, moved or removed by the user.

For the sake of this assignment, you can ignore handling errors (like illegal name or illegal parent).

The code below shows how the library is to be used. You need to implement the respective classes.

```
// Get access to the root directory
Folder *root = Root::Get();

// Create some folders inside root
// Arguments are: file name and parent
Folder *f_etc = new Folder("etc", root);
Folder *f_bin = new Folder("bin", root);
Folder *f_usr = new Folder("usr", root);
```

```
// List contents of root
// Should print (order and whitespaces can be different):
//   etc/ bin/ usr/
root->list();

// Create some files and sub-folders in /etc
Document *f_etc_foo = new Document("foo", f_etc);
Document *f_etc_bar = new Document("bar", f_etc);
Folder *f_etc_init = new Folder("init", f_etc);
Document *f_etc_init_cron = new Document("cron.conf", f_etc_init);

// List contents of /etc
// Should print (order and whitespaces can be different):
//    foo bar init/
f_etc->list();

// Delete a folder (and all its children)
delete f_etc_init;

// List contents of /etc
// Should print (order and whitespaces can be different):
//    foo bar
f_etc->list();

// Create a soft-link to /etc/foo in /usr
Softlink *f_etc_foo_dup = new Softlink("foo.ln", f_etc_foo, f_usr);

// List contents of /usr
// Should print:
//    foo.ln@
f_usr->list();

// Move a directory
f_usr->moveto(f_etc);

// List contents of /etc
// Should print (order and whitespaces can be different):
//    foo bar usr/
f_etc->list();

// List contents of /etc/usr
// Should print (order and whitespaces can be different):
//    foo.ln@
f_usr->list();
```

A reference design is provided below that you can use for implementation.

```
class Folder;

// Base class for all files
class File {
   public:
      // Constructor: takes file name and parent folder
      File(std::string name, Folder *parent);

      // ls command
      virtual void list() = 0;

      // mv command
      // new_name could be identical to current name
      // new_parent could be identical to current parent
      // Must erase self from current parent and insert self to new parent
      virtual void move(std::string new_name, Folder *new_parent);

      // Returns the name of the file
      virtual std::string name();

   protected:
      // ...

   private:
      // ...
};
```

Each file type is implemented as a derived class:

```
class Folder : public File
{
   public:
      Folder(std::string name, Folder *path);

      // Print file name, path and folder contents (file names) to std::cout
      virtual void list();

      // Move folder to a diffent name and/or path
      virtual void move(std::string new_name, Folder *new_path);

      // Folder names have a "/" suffix
      virtual std::string name() { return (File::name() + "/"); }

      // Delete the folder and all its contents
      ~Folder();
```

```
      // Additional helpers: Add an element to m_contents list
      void insert(File *new_child);

      // Additional helpers: Remove an existing element from m_contents
      void erase(File *new_child);

   private:
      // Folder contents
      std::list<File *> m_contents;
};

// Class "Root" represents the "/" root folder.
// There can be only one instance of root folder
class Root: public Folder
{
   public:
      static Root* GetInstance() {
         if (theInstance == NULL) {
            theInstance = new Root();
         }
         return theInstance;
      }

   private:
      // Root is its own parent
      Root() : Folder("/", this) { }

      // Can't move root
      virtual void move(std::string new_name, Folder *new_path);

      static Root* theInstance;
      // ...
};


class Document : public File
{
   public:
      Document(std::string name, Folder *path);

      // Print file name and path to std::cout
      virtual void list();

      // Move file to a diffent name and/or path
      virtual void move(std::string new_name, File *new_path);
```

```
        // Delete the file
        ~Document();
};

class Softlink : public File
{
   public:
        Document(std::string name, File *link_target, Folder *path);

        // Print file name and path to std::cout
        virtual void list();

        // Softlink names have a "@" suffix
        virtual std::string name() { return (File::name() + "@"); }

        // Move file to a diffent name and/or path
        virtual void move(std::string new_name, File *new_path);

        // Delete the link
        ~Softlink();
};
```

2.6. Technical Article

Write a technical article (in style of a blog). You can pick your own topics, giving a few below as suggestions. The article/blog should provide a good coverage on the topic. It can include code snippets, diagrams, general gotchas/pitfalls, user guidelines, etc. It should be original content - not copy - paste.

Topic suggestions:

- `const` in C++
- Use of `std::bind` (with an overview of function pointers - including pointers to member functions)
- Introduction to design patterns (pick some - visitor, factory, model-view-controller, etc)
- Introduction to smart pointers in C++
- Introduction to C++11 move semantics - rvalue and universal references, `std::move`, move constructors
- Introduction to C++11 features - Automatic type deduction using "auto" keyword, Enhancements to enum, Initializer lists, Explicitly defaulted and deleted special member functions and override specifier, Lambdas

You are strongly encouraged to write the article in AsciiDoctor (https://asciidoctor.org/) or Markdown (https://daringfireball.net/projects/markdown/) syntax - but if that is very distracting, you can use Google Docs.

2.7. Buttons on a Frame

Say you have a `class Button;` that shows a GUI button on screen. The `Button` class lets a client register a function that will be called when the user clicks the button.

```
// Contents of button.h
class Button
{
   public:
      typedef void (*fptr_t)(void);

      // Client-side function pointer to call when user clicks on button
      void on_click_handler(fptr_t f);

      // Called when the user clicks a button
      void click();
};
```

You should write a client class that, when instantiated, will create `N` such buttons (where `N` is a constructor time parameter).

```
// Contents of MyFrame.h
class MyFrame
{
   public:
      MyFrame(int N) : m_buttons(N), m_total_clicks(0), m_clicks(N, 0)
      {
         // ...
      }

      void print_stats()
      {
         std::cout << "Total Clicks: " << m_total_clicks << std::endl;
         for(int i=0; i<m_clicks.size(); ++i)
         {
            std::cout << " Button[" << i << "] clicks: " << m_clicks[i]
               << std::endl;
         }
      }

      std::vector<Button> m_buttons;
   public:
      int m_total_clicks;
      std::vector<int> m_clicks;
};
```

The requirement is the following: Each time the user clicks the `i'th` button, the `m_clicks[i]` and `m_total_clicks` should both get incremented. To achieve this, the class `MyFrame` will register a "click-handler" callback with each button.

It is acceptable to modify `Button` class if required - but it should remain generic.

A sample test-code is provided below:

```
int main()
{
   MyFrame f1(2);
   MyFrame f2(3);

   // Simulate few clicks for the two frames
   f1.m_buttons[0].click();
   f1.m_buttons[0].click();
   f1.m_buttons[0].click();
   f1.m_buttons[1].click();
   f1.m_buttons[1].click();

   f2.m_buttons[2].click();
   f2.m_buttons[2].click();
   f2.m_buttons[2].click();
   f2.m_buttons[1].click();

   // Should print:
   // Total Clicks: 5
   //  Button[0] clicks: 3
   //  Button[1] clicks: 2
   f1.print_stats();

   // Should print:
   // Total Clicks: 4
   //  Button[0] clicks: 0
   //  Button[1] clicks: 1
   //  Button[2] clicks: 3
   f2.print_stats();
}
```

2.8. LMC and Memory

2.8.1. LMC Overview

The Little Man Computer (LMC) is an instructional model of a computer, created by Dr. Stuart Madnick in 1965. LMC uses von-Neumann architecture. For simplicity, LMC uses decimal base and not binary.

LMC has 100 addressable locations (address can be `0` to `99`). It has two registers: an accumulator (`ACC`) and a program counter (`PC`). `PC` can hold an address - any value from `0` to `99`. `ACC` can hold data from `0` to `999`. LMC has a 1-bit single flag, `N` which is set when `ACC` becomes negative or overflows due to addition (as explained further below).

At reset, both registers and the flag `N` get set to `0`.

LMC has just 10 instructions - divided between arithmetic (`ADD`/`SUB`), load/store (`LDA`/`STA`), control-flow (`BRA`/`BRZ`/`BRP`/`HLT`) and I/O (`INP`/`OUT`).

The basic processing loop of LMC is:

- **Instruction Fetch**: The contents at current `PC` value is fetched from memory. `PC` is incremented by `1` after fetch.
- **Instruction Decode**: In this step, LMC decodes the instruction to determine the opcode and operand. `INP`, `OUT` and `HLT` have no operands. Remaining 7 instructions have a single operand.
- **Instruction Execute**: The instruction gets executed.

The instructions are described in the table below. Note that this assignment has a slightly modified specification of the LMC ISA (as found in Wikipedia). All instructions are 3-digit decimal numbers.

The contents of the memory at an address `xx` is denoted as `mem[xx]`.

Numeric code	Mnemonic code	Description	Pseudo-code

1xx	ADD	Adds contents of specified memory location to `ACC` (with overflow protection)	`if ((ACC + mem[xx]) > 999)`
			`  ACC := 999`
			`  N := 1`
			`else`
			`  ACC := ACC + mem[xx]`
			`  N := 0`

2xx	SUB	Subtracts contents of specified memory location from `ACC` (with underflow protection)	`if ((ACC - mem[xx]) < 0)`
			`  ACC := 0`
			`  N := 1`
			`else`
			`  ACC := ACC - mem[xx]`
			`  N := 0`

3xx	STA	Store the contents of `ACC` at specified location	`mem[xx] := ACC`
5xx	LDA	Load the contents from specified location to `ACC`	`ACC := mem[xx]`
6xx	BRA	Unconditional branch.	`PC := xx`
7xx	BRZ	Conditional branch (Branch If Zero).	`if (ACC == 0)` `  PC := xx`
8xx	BRP	Conditional branch (Branch If Positive).	`if (N == 0)` `  PC := xx`
901	INP	Read `ACC` from input device	`ACC := <input>`

Numeric code	Mnemonic code	Description	Pseudo-code
902	OUT	Write ACC to output device	<output> := ACC
0	HLT/COB	Stop working/end the program.	N.A.
N.A.	DAT	This is an assembler instruction which loads the value into the next available mailbox.	N.A

References:

- LMC overview: https://en.wikipedia.org/wiki/Little_man_computer

2.8.2. Modeling Requirements

In this assignment, you will implement a ISS (Instruction Set Simulator) of LMC. The memory will be separated from the ISS. A real processor will have an interface like AMBA AXI or AHB to access the memory. In this case, the ISS will have a abstract interface as described below.

The class `bus_if` abstracts the bus interface and allows single element reads and writes. Note that we use `uint8` for address - however, the legal values for address are only from `0` to `99`. Similarly, we use `uint16` for data, however the legal values are only from `0` to `999`.

```
// bus_if.h
class bus_if
{
   public:
      enum {BUS_OK, BUS_ERROR} status_t;
      virtual status_t read(uint8 addr, uint16 &data) = 0;
      virtual status_t write(uint8 addr, uint16 data) = 0;
};
```

The LMC class skeleton should be as follows:

```
// Files: LMC.h and LMC.cpp
class LMC
{
   public:
      bus_if *port; // Port to access memory

      // Resets the LMC: ACC, PC gets set to 0, flags cleared
      void reset();

      // Crunches the ISS forward by one "step": instruction fetch,
      // decode and execute
      void step();

      // Read from input device
      uint16 device_read()
      {
         uint16 data;
         std::cin >> data;
         return data;
      }

      // Write to output device
      void device_write(uint16 data)
      {
         std::cout << data;
      }
   private:
      uint8 m_PC;
      uint16 m_ACC;
      bool m_N;
};
```

As indicated, the model will use `port` for memory access - both instruction reads and load/stores. The `step` function will step the LMC forward by one instruction fetch/decode/execute cycle.

```
void LMC::step()
{

   port->read(m_PC, instruction);
   m_PC++;
   decode_and_execute(instruction);
}
```

The LMC will need a memory model to execute - this will be implemented as a separate class. The memory model implements the bus interface. The skeleton for the memory model is as shown below:

```
// Files: Memory.h and Memory.cpp
class Memory : public bus_if
{
   public:
      virtual status_t read(uint8 addr, uint16 &data);
      virtual status_t write(uint8 addr, uint16 data);

      // Load the memory with values as specified in file
      // Return "true" on success
      bool load(std::string file_name);
   private:
      // ...
};
```

The `load` method is to initialize the memory contents. The format of the file given as argument to the method is as follows:

- Each line of the file is either a comment, or an address, or memory content or a blank line
- Comments start with `#` in the first column of the row, addresses with an `A` and memory data with `D`.
- Spaces are not allowed, except as part of comment line.
- Comments are ignored from being processed further
- The data following an address line is stored at that address
- There can be several lines of data after an address line - these are placed contiguously

An example file is shown below:

```
# This is a comment line
# Blanks like below are ok

# Line below specifies the address
A 0

# Data follows. 0 is stored at 0
# 11 at 1, ..., 99 at 9
D 0
D 11
D 22
D 33
D 44
D 55
D 66
D 77
D 88
D 99

# Line below specifies a new address
A 20

# Data follows. 0 is stored at 20,
# 11 at 21
D 0
D 11
```

If there are any errors during load, the method prints the error and returns `false`. Execution should stop upon load failure.

Finally, the `main` function should be written as follows.

```
// File: main.cpp
// The executable should be run with the file to be loaded in memory as
// argument, like:
//   $ lmcsim memfile
int main(int argc, char *argv[])
{
   // TODO: Check if argc is right, etc..

   // Instantiate components
   LMC lmc;
   Memory memory;

   // "Connect" them
   lc3.port = &memory;

   // Initialize
   memory.load(argv[1]);

   // Simulate
   while(1)
   {
      lmc.step();
      // TODO: Add debug prints to monitor progress
   }
}
```

2.9. LC3 and Memory

References:

- LC3 ISA presentation: http://www.cs.utexas.edu/users/fussell/cs310h/lectures/Lecture_10-310h.pdf
- The LC-3 ISA reference: http://highered.mheducation.com/sites/dl/free/0072467509/104691/pat67509_appa.pdf
- LC3 Examples: http://people.cs.georgetown.edu/~squier/Teaching/HardwareFundamentals/LC3-trunk/docs/LC3-AssemblyManualAndExamples.pdf
- Wikipedia article: https://en.wikipedia.org/wiki/LC-3
- LC3 Assembler: https://github.com/davedennis/LC3-Assembler

2.9.1. LC3 Introduction

The LC-3 specifies a word size of 16 bits for its registers and uses a 16-bit addressable memory with maximum addressable memory of 2^{16} locations. The register file contains

eight registers, referred to as R0 through R7. All of the registers are general-purpose in that they may be freely used by any of the instructions that can write to the register file. There are other registers that influence the operation: 16-bit Program Counter (PC) and 16-bit Processor Status Register (PSR) - which includes 3 x 1-bit condition codes and other information.

Instructions are 16 bits wide and have 4-bit opcodes. The instruction set defines instructions for fifteen of the sixteen possible opcodes, though some instructions have more than one mode of operation. The architecture is a load-store architecture; values in memory must be brought into the register file before they can be operated upon.

All data in the LC-3 is assumed to be stored in a two's complement representation; there is no separate support for unsigned arithmetic. The LC-3 has no native support for floating-point numbers.

Arithmetic instructions available include addition (ADD), bitwise AND, and bitwise NOT, with the first two of these able to use both registers and sign-extended immediate values as operands. These operations are sufficient to implement a number of basic arithmetic operations, including subtraction (by negating values) and bitwise left shift (by using the addition instruction to multiply values by two). The LC-3 can also implement any bitwise logical function, because NOT and AND together are logically complete.

Memory accesses can be performed by computing addresses based on the current value of the program counter (PC) or a register in the register file; additionally, the LC-3 provides indirect loads and stores, which use a piece of data in memory as an address to load data from or store data to. Values in memory must be brought into the register file before they can be used as part of an arithmetic or logical operation.

The LC-3 provides both conditional and unconditional control flow instructions. Conditional branches are based on the arithmetic sign (negative, zero, or positive) of the last piece of data written into the register file. Unconditional branches may move execution to a location given by a register value or a PC-relative offset. Three instructions (JSR, JSRR, and TRAP) support the notion of subroutine calls by storing the address of the code calling the subroutine into a register before changing the value of the program counter. The LC-3 does not support the direct arithmetic comparison of two values. Computing the difference of two register values requires finding the negated equivalence of one register value and then, adding the negated number to the positive value in the second register. The difference of the two registers would be stored in one of the 8 registers available for the user.

2.9.2. Modeling Requirements

In this assignment, you will implement a ISS (Instruction Set Simulator) of LC-3, with following simplifications for the initial version of the ISA:

- Interrupts are not supported. RTI instruction is not implemented.
- Priority levels and Privileged modes are not implemented.

The memory will be separated from the ISS. The ISS will implement Harvard architecture, with separate access paths for instruction and data. A real processor will have an interface like AMBA AXI or AHB. In this case, the ISS will have a abstract interface as described below.

The class `bus_if` abstracts the bus interface and allows single element reads and writes.

```
// bus_if.h
class bus_if
{
   public:
      enum {BUS_OK, BUS_ERROR} status_t;
      virtual status_t read(uint16 addr, uint16 &data) = 0;
      virtual status_t write(uint16 addr, uint16 data) = 0;
};
```

The LC3 class skeleton should be as follows:

```
// Files: LC3.h and LC3.cpp
class LC3
{
   public:
      bus_if *iport; // Instruction access, only reads
      bus_if *dport; // Data access for load/store, reads and writes

      // Resets the LC3: R0-R7, PC gets set to 0, flags cleared
      void reset();

      // Crunches the ISS forward by one "step": instruction fetch,
      // decode and execute
      void step();
   private:
      std::vector<uint16> m_R;
      uint16 m_PC;

      // Various fields of PSR register
      bool m_privilege;
      int  m_priority;
      bool m_N;
      bool m_Z;
      bool m_P;
};
```

As indicated, the model will use `iport` for instruction access and `dport` for load/stores. The `step` function will step the state forward by one instruction fetch/decode/execute cycle.

```
void LC3::step()
{
   // ...
   iport->read(m_PC, instruction);
   decode_and_execute(instruction);
}
```

The LC3 will need a memory model to execute - this will be implemented as a separate class. The memory model implements the bus interface. The skeleton for the memory model is as shown below:

```
// Files: Memory.h and Memory.cpp
class Memory : public bus_if
{
   public:
      virtual status_t read(uint16 addr, uint16 &data);
      virtual status_t write(uint16 addr, uint16 data);

      // Load the memory with values as specified in file
      // Return "true" on success
      bool load(std::string file_name);
   private:
      // ...
};
```

The `load` method is to initialize the memory contents. The format of the file given as argument to the method is as follows:

- Each line of the file is either a comment, or an address, or memory content or a blank line
- Comments start with `#` in the first column of the row, addresses with an `A` and memory data with `D`.
- Spaces are not allowed, except as part of comment line.
- Comments are ignored from being processed further
- The data following an address line is stored at that address
- There can be several lines of data after an address line - these are placed contiguously

An example file is shown below:

```
# This is a comment line
# Blanks like below are ok

# Line below specifies the address
A 0x3000

# Data follows. 0x0000 is stored at 0x3000,
# 0x1111 at 0x3001, ..., 0x9999 at 0x3009
D 0x0000
D 0x1111
D 0x2222
D 0x3333
D 0x4444
D 0x5555
D 0x6666
D 0x7777
D 0x8888
D 0x9999

# Line below specifies a new address
A 0x1000

# Data follows. 0x0000 is stored at 0x1000,
# 0x1111 at 0x1001
D 0x0000
D 0x1111
```

If there are any errors during load, the method prints the error and returns `false`. Execution should stop upon load failure.

Finally, the `main` function should be written as follows.

```
// File: main.cpp
// The executable should be run with the file to be loaded in memory as
// argument, like:
//   $ lc3sim memfile
int main(int argc, char *argv[])
{
   // TODO: Check if argc is right, etc..

   // Instantiate components
   LC3 lc3;
   Memory memory;

   // "Connect" them
   lc3.iport = &memory;
   lc3.dport = &memory;

   // Initialize
   memory.load(argv[1]);

   // Simulate
   while(1)
   {
      lc3.step();
      // TODO: Add debug prints to monitor progress
   }
}
```

2.10. LC3 and Memory in SystemC

Reimplement the LC3 and memory model in SystemC. The following are the overview of changes to be done:

- `bus_if` should be an "interface proper". deriving from `sc_interface` class
- `LC3` and `Memory` classes should be SystemC modules, deriving from `sc_module`
- The `iport` and `dport` in `LC3` will be declared as SystemC ports of type `sc_port<bus_if>`
- The `step()` function in `LC3` will be converted to a `SC_THREAD`. After instruction fetch, insert a wait of 5 ns; and after decode/execute, insert an other wait of 5 ns.
- Rename `main` as `sc_main`. Modify the lines that set `iport` and `dport` to instead bind to `memory`. Remove the `while(1)` loop - and instead call `sc_start()`.

2.11. Add a Bus module to LC3

Extend the LC3 sub-system to include a switch/bus that can connect to arbitrary number of masters and slaves.

The bus should accept the number of masters and number of slaves as constructor parameters. Further, the bus model should allow the address map to be configured. Assume that the bus implements a "unified address map" (i.e., the address map is identical for all masters) and allows only a single contiguous address range per slave port.

You can use the below snippet as a skeleton.

```
class Bus : public sc_module, public bus_if
{
   public:
      Bus(sc_module_name n
            , int nMasters /* = number of slave ports in Bus */
            , int nSlaves  /* = number of master ports in Bus */);

      sc_vector<sc_port<bus_if>> mport;

      // 'port' gets associated to the address range from 'low_addr' to
      // 'high_addr'. Gets called multiple times, once for each slave.
      void add_address_range(uint16 low_addr, uint16 high_addr, int port);

      void end_of_elaboration()
      {
         // TODO: Check that all slaves have an address range. Check that
         // address range doesn't overlap. Check that low_addr <= high_addr
         //    --> else error out / sc_stop.
      }

      virtual status_t read(uint16 addr, uint16 &data)
      {
         /* TODO: Determine which port does "addr" map to. Say, it is "i" */
         return mport[i]->read(addr, data);
      }

      virtual status_t write(uint16 addr, uint16 data)
      {
         /* TODO: Determine which port does "addr" map to. Say, it is "i" */
         return mport[i]->write(addr, data);
      }
};
```

Create a top module that instantiates and connects `LC3`, `Bus` and `Memory`. Configure the bus to allocate the entire address range to the memory model. Complete the example with an `sc_main` that instantiates the top module.

```
class top : public sc_module
{
   public:
      top(sc_module_name n) : lc3("lc3"), memory("memory"), bus("bus", 2, 1)
      {
         lc3.iport(bus);
         lc3.dport(bus);

         bus.mport[0](memory);

         bus.add_address_range(0x0, 0xFFFF, 0);
      }
};
```

2.12. Simple combinational circuit

Implement an `AND` (2 inputs, 1 output), `OR` (2 inputs, 1 output) and a `NOT` (1 input, 1 output) gates in SystemC.

The input ports must be of type `sc_in<bool>` and outputs should be of type `sc_out<bool>`. The gate model should have a `SC_METHOD` that is statically sensitive to changes in the input port(s). This `SC_METHOD` should drive the output of the gate. The method must print the current simulation time and the current delta-cycle count whenever the output is driven. Use the skeleton below:

```
class AndGate : public sc_module
{
   public:
      sc_in<bool> in0;
      sc_in<bool> in1;
      sc_out<bool> out;

      SC_CTOR(AndGate) {
         // Declare method process, sensitivity, etc
      }

      void end_of_elaboration() {
         // Initialize outputs
      }

      void method() {
         std::cout << "AndGate: t=" << sc_time_stamp() << ", delta=" <<
sc_delta_count() << std::endl;
         // TODO: Also print inputs and the value that will be driven out
         // TODO: Drive the output
      }
};
```

Use the gates to create a hierarchical model that takes 4 inputs (A, B, C, D) and provides a single output as per the following truth table:

	A	B	C	D	Output
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1

	A	B	C	D	Output
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

TIP

After K-map simplification, the output can be expressed as the following function of inputs:

```
Output = A.(!C) + A.(!B) + B.C.(!D)
```

Use the following skeleton for the hierarchical model:

```
class hm : public sc_module
{
   public:
      sc_in<bool> in_A;
      sc_in<bool> in_B;
      sc_in<bool> in_C;
      sc_in<bool> in_D;
      sc_out<bool> out;

      SC_CTOR(hm) {
         // Connect child instances
      }
      // TODO: Declare gates as child instances
};
```

Create a test module that will drive the inputs and observe the output. Use the following skeleton for the test module:

```
class hm_test : public sc_module
{
   public:
      sc_out<bool> out_A;
      sc_out<bool> out_B;
      sc_out<bool> out_C;
      sc_out<bool> out_D;
      sc_in<bool> in;

      SC_CTOR(hm) {
         SC_THREAD(test);
      }
      // TODO: Implement end_of_elaboration, other required functions

      void test() {
         // TODO: Call stimulate_n_monitor() with various patterns
      }

      void stimulate_n_monitor(bool a, bool b, bool c, bool d) {
         out_A = a; out_B = b; out_C = c; out_D = d;
         std::cout << "HM test: A: " << a << ", B: " << b
            << ", C:" << c << ", D:" << d << std::endl;
         wait(1, SC_NS);
         std::cout << "HM test: Result: " << in << std::endl;
         wait(1, SC_NS);
      }
};
```

Create a top-level model that instantiates the `hm` and `hm_test` and connects them. For connecting an `sc_out` port to `sc_in` use an `sc_signal`.

Follow-up activities after completing the assignment:

- Try to reason out the prints - are delta-cycle counts and process execution as expected?
- Use `sc_buffer` instead of `sc_signal`. Are the prints are as expected?
- Change the methods in the gate module to use dynamic sensitivity using `next_trigger`. This should result in exactly the same prints as earlier.
- Change the `SC_METHOD` in the gate module to an `SC_THREAD` instead. Use dynamic sensitivity. Again, the prints must be exactly same as earlier.
- Use `sc_trace` to trace the ports and signals and view the resulant VCD file with a viewer like `gtkwave`. Are you able to see the "delta-cycle" changes?

2.13. Game of Life - I

Conway's game of life is a very interesting 0-person game / cellular automation. More details can be found on the Wikipedia page: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

The objective of this exercise is to use SystemC to model the game. Each cell should be modelled as a SystemC module: the cells are connected to their neighbors using ports - that happens in an another top-level SystemC module. All the cells have a clock input that are driven by a single clock source. A clock process executes in each cell that decides the state for the next generation.

The first version of this assignment requires using a `sc_port` of your own interface. Each cell implements this interface - as well as connects to other cells using ports of this interface.

```
// cell.h / cell.cpp
class cell_if : public sc_interface
{
   public:
      // Return true if living, false otherwise
      virtual bool is_alive() = 0;
};

class cell : public sc_module, public cell_if
{
   public:
      sc_in<bool> clk;
      sc_vector<sc_port<cell_if>> neighbor;

      void step(void)
      {
         while(1) {
            wait(clk.pos());
            // TODO: Update state based on neighbors' current state
         }
      }

      bool is_alive()
      {
         // TODO: Provide current state to neighbors
      }

      // TODO: Implement constructor and other functions
};

// top.h / top.cpp
class top : public sc_module
{
   // TODO: Instantiate a grid of cells and connect them

};
```

Adapt the code snippet above suitably for completing the assignment. Note in particular that the *next* state of a cell depends on *current* state of its neighbors (along with its own current state).

Also consider:

- What are the strategies for accomodating the corner-cases, where a cell may have fewer than 8 neighbors?

- How will you handle the two-dimensional aspect of the problem. For example, you will need a 2D array of `cell` modules. Can you create an `sc_vector<sc_vector<cell>>` ? If not with `sc_vector`, how will you manage the cell array?

2.14. Game of Life - II

Redo the assignment to instead use `sc_in/sc_out` ports directly.

```
// cell.h / cell.cpp
class cell : public sc_module
{
   public:
      sc_in<bool> clk;
      sc_vector<sc_in<bool>> neighbor;
      sc_out<bool> state; // true = alive, false = dead

      void step(void)
      {
         while(1) {
            wait(clk.pos());
            // TODO: Update state based on neighbors' current state
         }
      }

      // TODO: Implement constructor and other functions
};

// top.h / top.cpp
class top : public sc_module
{
   // TODO: Instantiate a grid of cells and connect them

};
```

2.15. Use of sc_export

Modify the "Game of Life - I" to use `sc_export` construct:

```
class cell : public sc_module, public cell_if
{
   public:
      sc_in<bool> clk;
      sc_vector<sc_port<cell_if>> neighbor;
      sc_export<cell_if> state; // Note

      // ...

};
```

2.16. Use of Initiator/Target Sockets

Modify the LC3 + Bus + Memory sub-system to use `tlm_initiator_socket` and `tlm_target_socket` (and eliminate the abstract class `bus_if` completely). Implement only the blocking transport `b_transport` (`nb_transport` and debug/DMI related calls can be dummy). The `BUSWIDTH` parameter for the ports can be set to `32`. Use only single element 16-bit transfers (so data length and streaming width is set to `2`). Byte enables are always set for both bytes. Only the lower 16 bits of address is used.

Modify the memory to insert a delay for reads and writes done via `b_transport`.

2.17. Adding Interrupts for LC3

In this assignment, we will add support for interrupts in LC3. Note that modifications done to LC3 in this assignment deviates from the official LC3 specifications.

2.17.1. ARM Interrupt Architecture

We base the modifications to LC3 on ARM interrupt architecture.

ARM defines the following exception table:

Exception	Address
Reset	`0x00`
Undefined Instruction	`0x04`
Software Interrupt	`0x08`
Pre-fetch Abort	`0x0C`
Data Abort	`0x10`
Reserved	`0x14`
IRQ	`0x18`

Exception	Address
FIQ	0x1C

When ARM encounters an exception - for example, upon receiving an interrupt, the processor does the following steps:

- Save the current PC value on stack
- Set flags to indicate mode of operation
- Set PC to exception address, and continue execution

The S/W present at exception address is then responsible for handling the exception. Since the exception table only provides 4 bytes for implementing the handler (for all exceptions except FIQ), those 4 bytes usually have an unconditional jump to the real exception handler. Upon returning from exception, the reverse process is done to ensure execution continues as expected.

ARM also provides "mode-specific banking" of some of the general-purpose registers: for example, R13 (stack pointer) and R14 (link register). This provides a private stack for each mode.

ARM supports exception prioritization and nested handling of exception.

For further reading, check ARM7 datasheet (this is one of the simplest ARM cores - more recent ARM cores have a slightly evolved version of the interrupt architecture): http://infocenter.arm.com/help/topic/com.arm.doc.ddi0086b/DDI0086B_710t_ds.pdf

2.17.2. LC3 Interrupt Modeling Requirements

All the changes below should be done to the SystemC model of LC3 that uses TLM sockets.

In case of LC3, as a part of this assignment, you will implement:

- Exception vectors for reset and IRQ
- No mode-banked registers
- No priority levels
- Only a single privilege level

The exception vector table to be implemented is as follows:

Exception	Address
Reset	0x00
IRQ	0x01
Reserved	0x02

Add the following signals to the LC3:

- sc_in<bool> nReset; → this is reset input. Upon reset, current execution is aborted, all registers are set to default value, PC is set to the exception address for reset (i.e., 0x00). Note that nReset is ACTIVE LOW signal - which means if the value on the signal is 0, reset should happen.
- sc_in<bool> nIRQ; → this is interrupt input (also ACTIVE LOW). This signal is checked in each step just before instruction fetch. If it asserted (nIRQ == 0), then the current value of PC is pushed on stack, PC is set to the exception address (0x01). R6 register is used as stack pointer.

When the RTI instruction is executed, the processor should pop the top of stack, store the value in PC and continue execution. When the processor is servicing the interrupt (i.e., after jumping to the IRQ exception address and until RTI is executed), it should ignore the value of nIRQ.

Since the available space at exception address is just 1 word long, there should be a branch present at that location to the real handler. For this assignment, the reset handler should be placed at 0x10, and the IRQ handler at 0x20.

So the instruction at 0x00 should be:

```
BR 0x10; // Machine code: 0000-1110-0001-0000
```

The instruction at 0x01 should be:

```
BR 0x20; // Machine code: 0000-1110-0010-0000
```

The reset handler should initialize the stack pointer (R6) to some meaningful value. We will use 0x100 for this assignment. (Question: what will happen if an interrupt is received just after reset, before the reset handler has initialized R6?). After that, the reset handler should branch to the real application location (say 0x3000).

The IRQ handler - for this assignment - can be dummy loop that keeps the core busy for few cycles. The handler should finally return with RTI.

After implementing the changes, connect the IRQ and reset ports to sc_signal objects in the top-level class / sc_main. Set the signal values to true before calling sc_start (so signals are considered de-asserted). By calling sc_start with a specified time parameter, and controlling the value of the signals between successive invocation of sc_start, test the implemented functionality.

```
int main(int argc, char *argv[])
{
   // ...
   sc_signal<bool> nReset;
   sc_signal<bool> nIRQ;

   lc3.nReset(nReset);
   lc3.nIRQ(nIRQ);

   // De-assert:
   nReset = true;
   nIRQ = true;

   sc_start(100, SC_NS);

   // Assert reset
   nReset = false;
   sc_start(40, SC_NS); // Check if reset occurred

   // De-assert reset
   nReset = true;
   sc_start(50, SC_NS); // Ensure enough time - at least stack pointer should
be setup

   // Assert interrupt
   nIRQ = false;
   sc_start(50, SC_NS); // Check if the IRQ handler executed

   // De-assert interrupt
   nIRQ = true;
   sc_start(300, SC_NS); // Check if execution continues after IRQ handler
returns
}
```

2.18. Add simple UART to LC3 sub-system

TODO.

2.19. Add simple DMAC to LC3 sub-system

TODO.