Computer Simulation using Object-oriented design in C

Nicholas A. Hays, Henry N. Lawrence, Benjamin J. Pasero

University of Washington – Tacoma

February 21, 2016

Table of Contents

[Abstract 3](#_Toc445900023)

[Overview 4](#_Toc445900024)

[Assembler 5](#_Toc445900025)

[Assembler 5](#_Toc445900026)

[Graphic User Interface 6](#_Toc445900027)

[Simulator 7](#_Toc445900028)

[View 8](#_Toc445900029)

[How it works 9](#_Toc445900030)

[CPU 10](#_Toc445900031)

[Register File 11](#_Toc445900032)

[Memory 12](#_Toc445900033)

[ALU 13](#_Toc445900034)

[Control Unit 14](#_Toc445900035)

[Appendix A 17](#_Toc445900036)

[LC-2200 ISA Format 17](#_Toc445900037)

[Simulator Flow Diagram 21](#_Toc445900038)

[Simulator Component Hierarchy 22](#_Toc445900039)

[Data Path Design 23](#_Toc445900040)

[Appendix B 24](#_Toc445900041)

[Assembler docs 24](#_Toc445900042)

[Assembler GUI docs 27](#_Toc445900043)

[Appendix C 30](#_Toc445900044)

[Simulator Main docs 30](#_Toc445900045)

[List docs 31](#_Toc445900046)

[MenuBar docs 32](#_Toc445900047)

[Appendix D 33](#_Toc445900048)

[Control Unit docs 33](#_Toc445900049)

[Register File docs 35](#_Toc445900050)

[Memory docs 37](#_Toc445900051)

[ALU docs 39](#_Toc445900052)

[LC-2200 UML 42](#_Toc445900053)

# Abstract

This project is an abstraction of a computer’s data path design using object-oriented programming in C with supporting GIMP Toolkit (GTK+ 3) libraries to simulate the fetch-decode-execute cycle of an assembled program using the LC-2200 Instruction Set Architecture (ISA). The LC-2200 ISA supports RISC-style instructions with four types of addressing modes (R-type, I-type, J-type, and O-type) and eight different instructions: add, nand, addi, lw, sw, beq, jalr, and nop. The register-oriented 32-bit single-bus data path design contains five main components to support the CPU: memory, ALU, register file, program counter (PC), and instruction register (IR). These components serve as vehicles to carry out the simple instructions specified in the LC-2200 ISA to cater the needs of any high-level language.

*Keywords:* gtk, c, object-oriented, PC, ALU, register, memory, assembler

# Overview

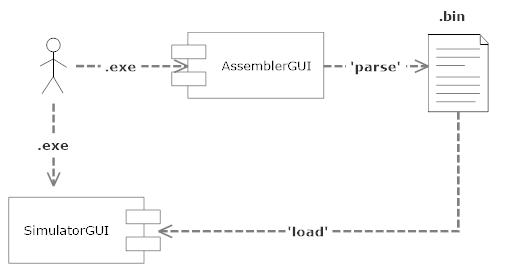
There are several systems involved when simulating the execution of a computer program in a processor. Before the process begins, however, the instruction set must be defined in order to establish the interface between the software and the hardware. The LC-2200 ISA will serve as the link between the assembly program and the instruction execution. To initiate the process, a set of instructions are inserted into the assembler GUI using the instruction format specified in the ISA (refer to Appendix A to see a comprehensive list of the instruction’s format supported by LC-2200 ISA). Once the program is complete, the assembler will assemble the program into its binary representation. The binary file will contain the bit representation of the instructions needed by the simulator to execute the program. The simulator will then load the binary file into its GUI to reflect the instructions located in the CPU’s memory. The CPU will then execute each instruction when signaled to do so. After each instruction execution, the simulator will update the changes made to each of the register values along with the Program Counter (PC) and Instruction Register (IR). This process will continue until all instructions have been executed (see Figure 1).

Figure 1: Program Flow

# Assembler

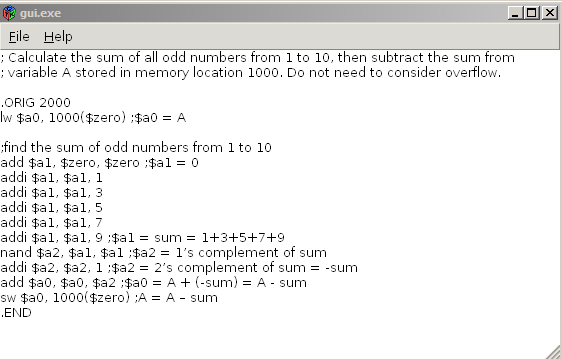
## Assembler

At the heart of the entire assembler program is the assembler (assembler.c) itself, which translates strings of assembly code into machine code for the simulator to execute. This is all done through a call to CreateMachineCode which takes an instruction string as an argument and returns an integer representing the machine code for that instruction. The CreateMachineCode function begins by setting the opcode of the output machine code. This is done in the getOpcode function which returns a machine language integer with only the bits for the opcode set. String comparisons are used to determine the correct opcode for an instruction.

Next, the rest of the bits in the output machine code are set based on the instruction type. The getInstructionType method returns the type of an instruction, either R, I, J or O, based on the opcode of the instruction machine code. The opcode is bit shifted into a regular integer and opcode value ranges help easily determine the instruction type. Based on the instruction type, the rest of the instruction string is parsed and machine code values are set accordingly. Register bits are set using the setRegister() function. The function takes a char argument representing which register bits in the machine code binary to set, x, y or z. A call to getRegNum returns the number of a register based on its string representation. The bits to set are shifted according to which register is be set and a bitwise OR combines the machine code so far with the register value. Setting an immediate value works in almost exactly the same way. The .ORIG directive is handled by simply returning the origin address. It is assumed the .ORIG directive will be the first statement in a program. When reading a file, the simulator will read the first integer as the origin address. The .END directive is interpreted as a halt instruction.

## Graphic User Interface

The Graphic User Interface (GUI) for the assembler (assemblerGUI.h) was created using the GTK widget toolkit and designed using the Glade user interface designer. The GUI gives the user the option to load in a text file to the assembler, save the current text, or enter or clear the current text. Execution begins with the startGTK function. In the function the XML design of the user interface, created with Glade, is loaded and all the appropriate signals for menu item clicks are linked to their corresponding functions.



*Figure 2:* Assembler GUI

All the callback functions in assemblerGUI.h are executed on menu item clicks. The “clear menu item” simply references a text buffer and clears every line. The GTKTextBuffer object is important to the GUI, it is what allows the text in the TextView (seen by the user) to be accessed and manipulated. The save menu item’s callback function is saveActivate. When called, a file chooser dialog is displayed. If the user chooses a file and doesn’t click cancel, each line of the text buffer is written to the selected text file. The open menu item and its callback openActivate work very similarly. Interaction with assembler.c, specifically createMachineCode, happens in assembleBuffer. After assembleActivate is called when a user clicks the assemble menu item the buffer is assembled in assembleBuffer. The buffer is iterated through and after each line is assembled to machine code it is written to a binary file of the user’s choosing. Comments and blank lines are avoided with calls to isValidLine.

# Simulator

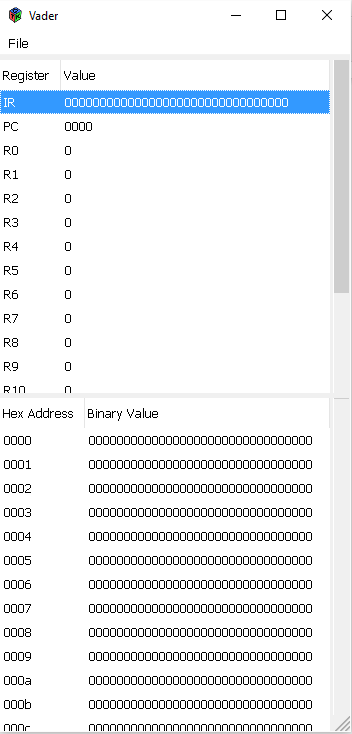
The simulator is a GUI that coordinates the display of the currently executing program by gathering data from the CPU, structuring it into a list model, and rendering it onto the screen. The simulator has two data models—instruction and register— each of which has respective widgets that reflect the program’s current values at a particular memory address and the current register value of all the GPR’s, IR, and PC . When the user launches the simulator executable, the startSimulator function begins to construct the GUI components (see Simulator Component Hierarchy in Appendix A for details). When the user loads a program into the simulator, the simulator will do several things: (1) begin to read the assembled file’s bytes into the CPU memory, (2) set the list store data, and (3) render the data in the view. The simulator responds to key event signals namely ‘F5’ initiated by the user to execute each instruction of the program (see Simulator Flow Diagram in Appendix A for details). The simulation will end when all instructions have been executed.

## View

The simulator’s GUI is built entirely with GTK+ 3 widgets without the support from Glade’s user interface designer (see Figure 3 below). The top level component is a GtkWindow which uses a GtkBox layout manager to organize subcomponents. The GtkBox lays out components by stacking them on top of one another in a vertical fashion. The first component in the GtkBox is a GtkMenuBar. The GtkMenuBar contains a File menu which contains a list of menu items. The first menu item, ‘Load Program’, launches a GtkFileChooser widget which prompts the user to import the assembled file into the simulator.

The next file menu item, ‘Reinitialize Machine’, is deprecated, but will be active in the next version. Its purpose is to clear memory in control unit, clear registers in the register file, reset the PC and IR, call create\_instruction\_view\_and\_model and create\_register\_view\_and\_model functions to clear the stores, and call refresh\_list() to update data in the view (refer to Appendix C for simulator documentation).

The last file menu item, ‘Exit Program’, terminates all resources attached to the simulator. The next two components in the GtkBox are both GtkScrolledWindow’s. The first scrolling window is a container for the register’s GtkTreeView widget, and the second is a container for the instruction’s GtkTreeView widget. The need to separate the instruction view from the register view becomes evident when running the simulation.



*Figure 3:* Simulator GUI

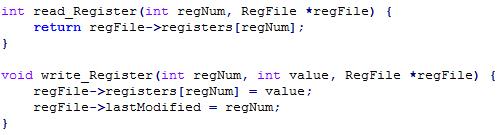
## How it works

The simulator is able to coordinate the display of its GUI to reflect the state of the CPU through various helper functions and supporting data models. When the assembled program is loaded into the simulator, the .orig address is retrieved, and the transfer of data (in 4 byte blocks) from the file into the CPU’s memory begins. After each four byte block is read, a call to the refresh\_lists function is generated. During this function call, the CPU’s memory is queried for its last modified address. This address will supply a gtk\_tree\_iter object with a reference to the location to write in the instruction list store. If the location is valid, a call to the gtk\_list\_store\_set function is made with the appropriate arguments, and the instruction model data is now synchronized with the memory in the CPU. This process repeats itself for registry data when the simulation begins; however, both views must be rendered accordingly to reflect the changes made in the data models. Each view’s column renders its cells with ‘text’ data, and therefore requires helper functions to transfer integer addresses and registry values into binary strings (refer to Helper Functions in Appendix C for details). Once the binary strings are constructed, the cell’s data can now be rendered appropriately to the GtkTreeView. After the last 4 byte block has been read from the file into the simulator, the process of executing the instructions begins. ‘F5’ key-press events generated by the user will signal the advance\_line callback function. The advance\_line function will immediately request the CPU to execute the next instruction, and refresh the lists in the exact manner described above. This process will terminate when all instructions have been executed.

# CPU

**LC-2200 Design**

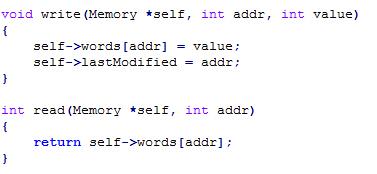
The design we used for our LC-2200 simulator consists of four main components: a register file containing sixteen register values, a memory unit containing a variable number address spaces, an ALU designed for the basic necessary mathematical functions, and a control unit which contains all previously mentioned components and oversees the process of fetching, decoding, and executing instructions (see Data Path Design in Appendix A for details). Each component is created using object oriented C classes (refer to Appendix D for the UML diagram). This implementation of a LC-2200 machine will execute any standard LC-2200 instruction.



*Figure 4: Register file code snippet.*

## Register File

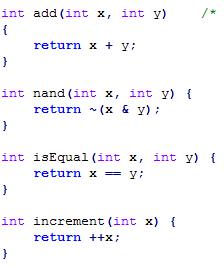
The register file contains the data and functions that simulate the register file in a real CPU. An array is used to store all register values. The number of the register in machine code is the same as the index in the array where its value is stored. This method allows for easier parsing of operands when executing instructions. The purpose of each register is the same as the LC-2200 standard shown in Appendix A. For example, in the registers array the integer value at index 0 corresponds to the $zero register and will always have a value of 0. The register file also contains two public methods, shown in figure 4, and two fields. The first method is a register read method that returns the value stored in the specified register. The second method is a write method that writes a specified value to a register. The register file also contains a last modified field which contains the register number of the most recently modified register. This was added in order for the simulator GUI to more efficiently update its list of registers by only rendering the most recently modified register rather than re-rendering the entire list every time.



*Figure 5:* Memory code snippet.

## Memory

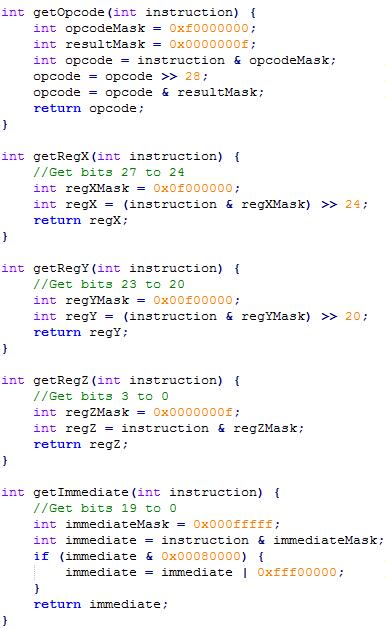
The memory class simulates the main memory in a CPU or RAM. This is implemented in a similar fashion as the register file. Memory is simulated using an array of integers. The address a value would be stored in memory is the same as the index it is stored in the array. To read or write a value at an address you must simply use the memory address as the index in the array. Again just like the Register File the memory class has two public functions shown in figure 5. The write method will write a value to an address in the simulator memory. The read method will return a value located in a specific address in memory. Memory also has a last modified field which indicates the address of the last write to memory. This is was added so the GUI would only re-render a single cell rather than the whole list.



*Figure 6:* ALU code snippet.

## ALU

The ALU simulates the ALU unit in a CPU. The ALU performs all the required mathematical functions needed in the CPU. The ALU has no modifiable fields, and contains four function pointers. The first function being ADD, which takes two integer parameters and returns their sum. The second is the NAND function which also takes two integer parameters. The NAND function performs both an AND NOT operation with the parameters and returns the result. The third function is called isEqual() which also takes two integer parameters. This function compares the two integers and returns a value of 1 if they are equal in value, or a 0 if they are not. The final function pointer in the ALU is name increment. The increment function takes one parameter and returns the addition of the value and one. Though this function is easily performs in C, we decided to add this to the ALU class in order to more accurately match the actual operations of a CPU which would use the ALU to increment a value.



*Figure 7:* Control Unit code snippet.

## Control Unit

The control unit is an implementation of a finite state machine and controls the flow of the data path. It contains a single register file, memory, and ALU which are all publicly accessible. It also contains a program counter and instruction register which it uses to store the address of the next instruction and the current instruction being executed respectively. There is only one public function in the control unit named nextInst. This function initiates the process to fetch, decode, and execute the next instruction in the program.

During the fetch phase the control unit retrieves the instruction in the memory address specified in the program counter. The retrieved instruction is stored in the instruction register and the program counter is then incremented.

Next, the control unit enters the decode phase. The instruction stored in the instruction register is parsed for the opcode and operands. To parse the integer the control unit uses bit masking. For example, to retrieve the op code we AND the instruction with the hexadecimal integer 0xf0000000 and then bit shift the result 28 bits to the right, leaving us with the integer value of the first four bits of the instruction. These bits make up the opcode for the instruction. Because we do not know the op code during the decoding phase we simply extract all possible combinations of data needed. Codes for register ‘x’, register ‘y’, register ‘z’, and immediate value are all extracted from the instruction regardless of op code. Instruction formats can be viewed in Appendix A. The opcode and operands decoded are then passed to the third and final phase of the data path

The execute phase of the data path performs the action specified in the instruction. Using the op code retrieved during the decode phase the control unit will use the values retrieved to carry out the instruction. For example, if the op code is that of an ADDI instruction the control unit will read the value of the ‘y’ register specified in the instruction. Using the ALU, it will then add the value from that register to the immediate value retrieved during decoding. The control unit will then store the result in the x register specified by the instruction. The instruction has now been performed and the control unit waits for the nextInst function to be called again where it will repeat this process with the next instruction specified in the program counter.

# Appendix A

## LC-2200 ISA Format

-----------------------------------------------------------------

Instruction Formats

-----------------------------------------------------------------

There are five instruction formats. Bit 0 is the least-significant:

R-type instructions (add,nand):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-4: unused (should be all 0s)

bits 3-0: reg DST

I-type instructions (addi, lw, sw, beq):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-0: OFFSET (a 20-bit, 2s complement number with a range

of -524288 to +524287

J-type instructions (jalr):

bits 31-28: opcode

bits 27-24: reg A

bits 23-20: reg B

bits 19-0: unused (should be all 0s)

O-type instructions (halt, ei, di, reti):

bits 31-28: opcode

bits 27-0: unused (should be all 0s)

-----------------------------------------------------------------

Register Convention

-----------------------------------------------------------------

Registers indicated with a '$' sign. The register names in assembly

are according to their use in the assembly convention:

regno name use callee-save

----- ---- ------------------------- -----------

0 $zero always zero (by hardware) n.a.

1 $at reserved for assembler n.a.

2 $v0 return value no

3 $a0 argument or temporary no

4 $a1 argument or temporary no

5 $a2 argument or temporary no

6 $a3 argument or temporary no

7 $a4 argument or temporary no

8 $s0 saved register YES

9 $s1 saved register YES

10 $s2 saved register YES

11 $s3 saved register YES

12 $k0 reserved for OS/traps n.a.

13 $sp stack pointer YES

14 $fp frame pointer YES

15 $ra return address YES

-----------------------------------------------------------------

Instruction Semantics

-----------------------------------------------------------------

Assembly language Opcode in binary Action

name for instruction (bits 31/30/29/28)

-----------------------------------------------------------------

add (R-type format) 0000 add contents of A with

ex: add $v0, $a0, $a1 contents of B, store results in

DST. Ex: $v0 := $a0 + $a1

nand (R-type format) 0001 nand contents of A with

ex: nand $v0, $a0, $a1 contents of B, store results in

DST. Ex: $v0 := ~($a0 + $a1)

addi (I-type format) 0010 Add OFFSET to the contents of A

ex: addi $v0, $a0, 25 and store the result in B.

Ex: $v0 := $a0 + 25

lw (I-type format) 0011 load B from memory. The memory

ex: lw $v0, 0x42($fp) address is formed by adding

OFFSET to the contents of A.

Ex: $v0 := memory[$fp + 0x42]

sw (I-type format) 0100 store B into memory. The memory

ex: sw $a0, 0x42($fp) address is formed by adding

OFFSET to the contents of A.

Ex: memory[$fp + 0x42] := $a0

beq (I-type format) 0101 compare the contents of A and B.

ex: beq $a0, $a1, done If they are the same, then

branch to the address

PC+1+OFFSET, where PC is the

address of the beq instruction.

Ex: if ($a0 == $a1)

PC := (PC+1)+OFFSET

\*\*\* NOTE \*\*\*

For programmer convenience (and

implementor confusion), the

assembler \*computes\* the OFFSET

value from the number or symbol

given in the instruction and the

assemblers idea of the PC. In the

example, the assembler stores

done-(PC+1) in OFFSET so that

the machine will branch to label

"done" at run time.

jalr (J-type format) 0110 First store PC+1 into B,

ex: jalr $at, $ra where PC is the address of the

jalr instruction. Then branch to

the address now contained in A.

Note that if A is the same as B,

the processor will first store

PC+1 into that register, then end

up branching to PC+1.

Ex: $ra := PC+1; PC := $a0

halt (O-type format) 0111 halt the machine: i.e. do nothing

ex: halt and let the simulator notice that

the machine halted.

ei (O-type format) 1010 enable interrupts

ex: ei

di (O-type format) 1011 disable interrupts

ex: di

reti (O-type format) 1100 return from interrupt by loading address

ex: reti stored in $k0 into the PC and then

enabling interrupts

----Assembler Directives----

noop (pseudo-op) n.a. No operation: does nothing (actually

ex: noop Emits "add $zero, $zero, $zero")

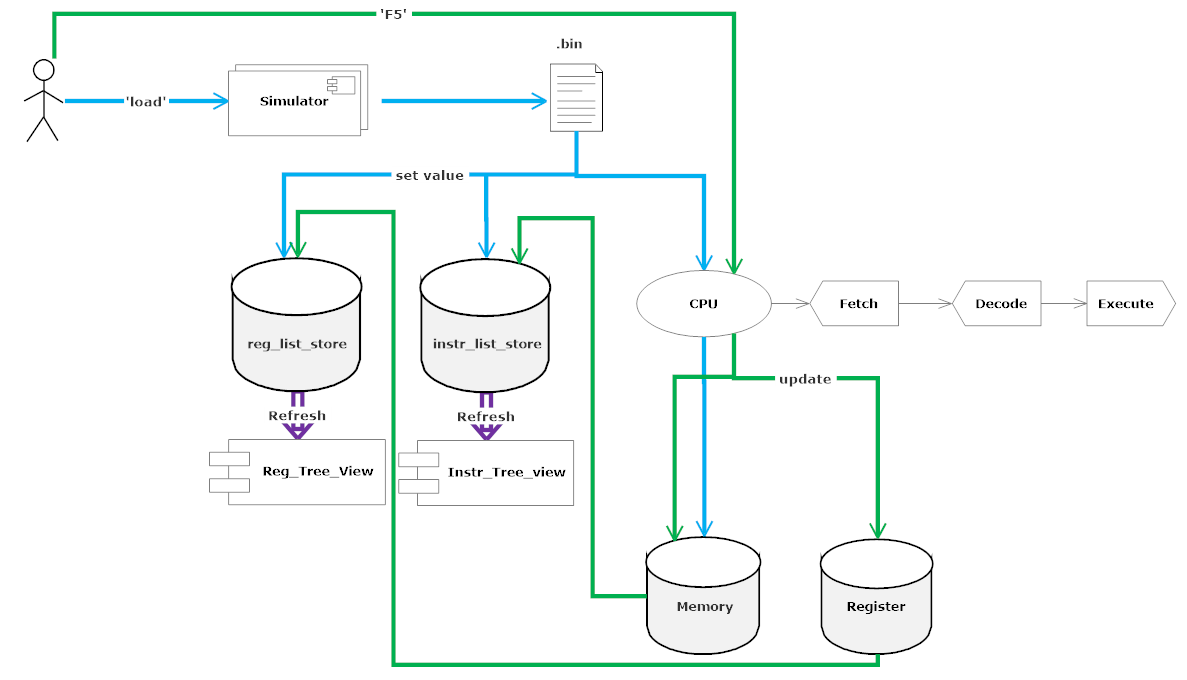
.word (pseudo-op) n.a. fill word with a value.

ex: .word 32 Ex: fill the current location

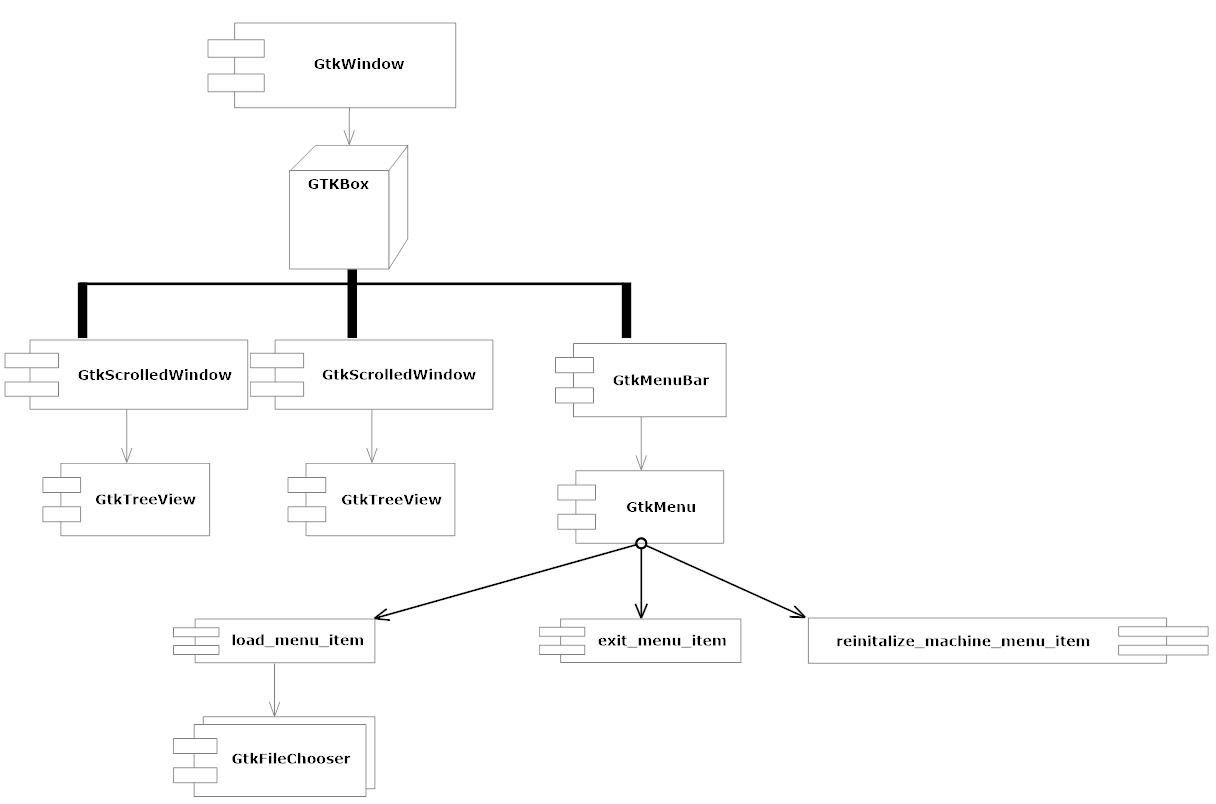
with the 32-bit representation of

the number "32"

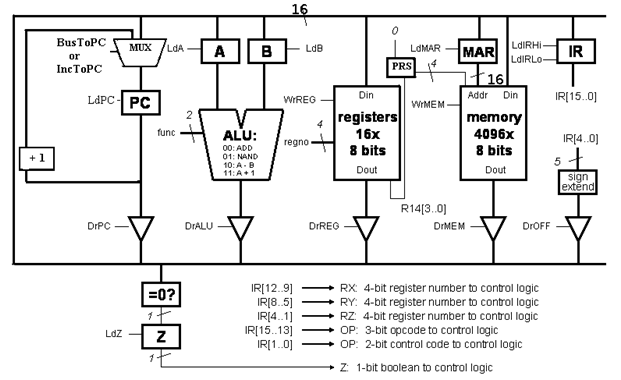
## Simulator Flow Diagram



## Simulator Component Hierarchy

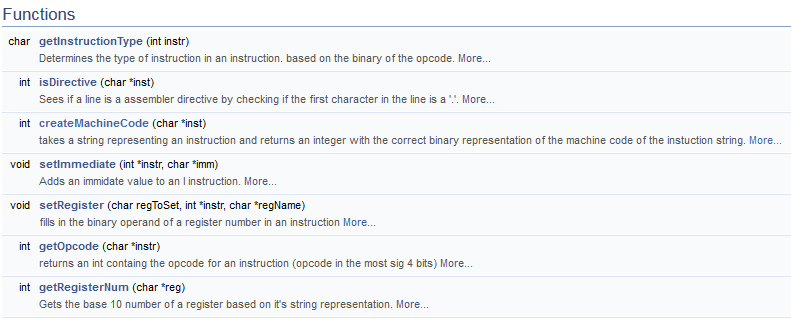


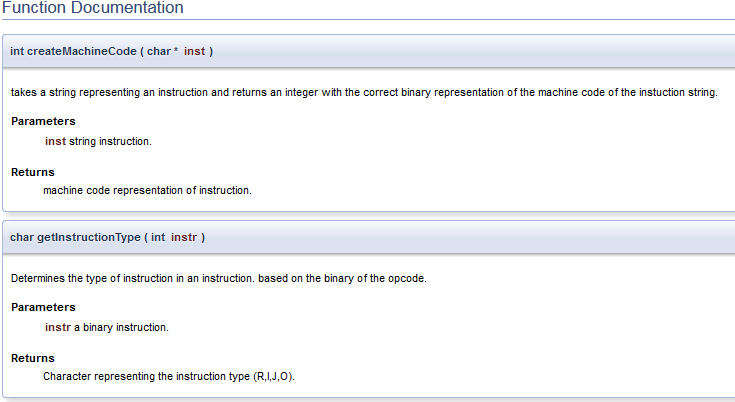
## Data Path Design



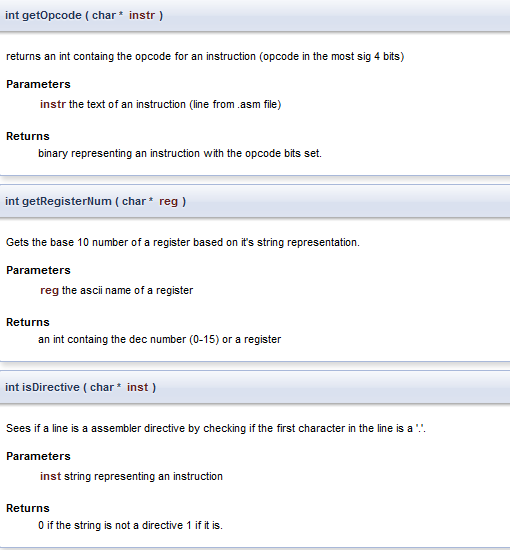
# Appendix B

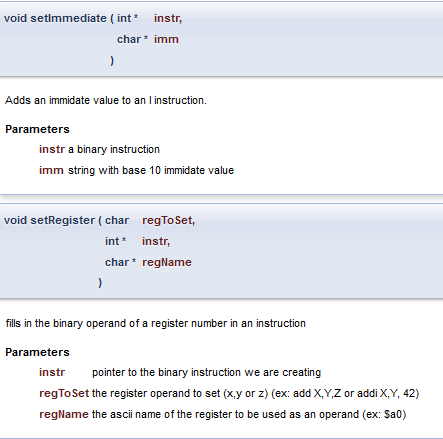
## Assembler docs





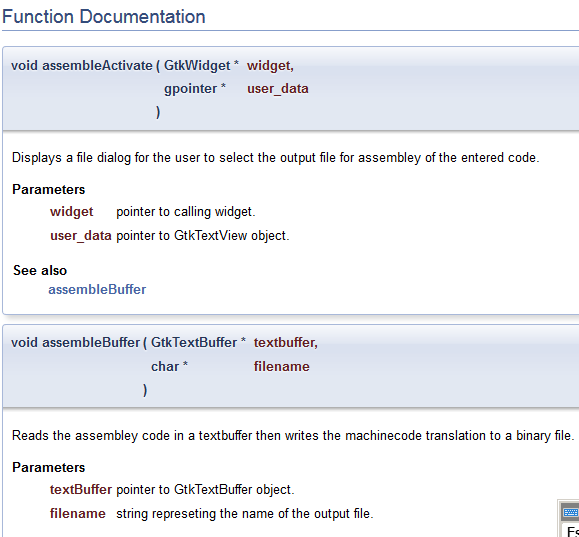
# 



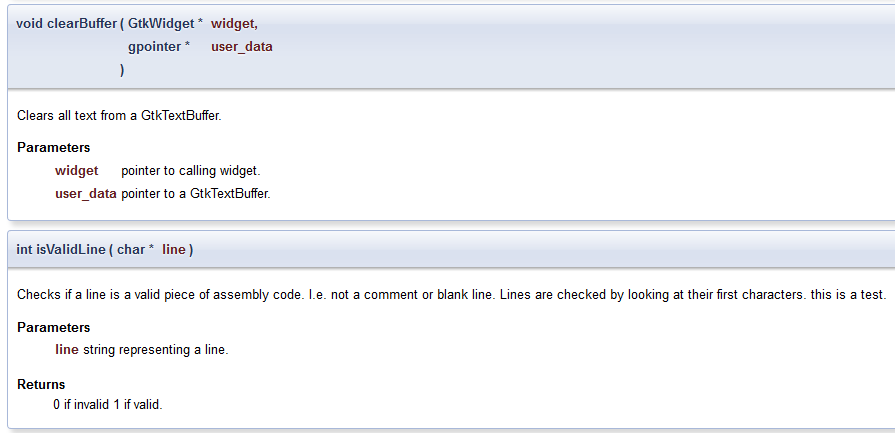


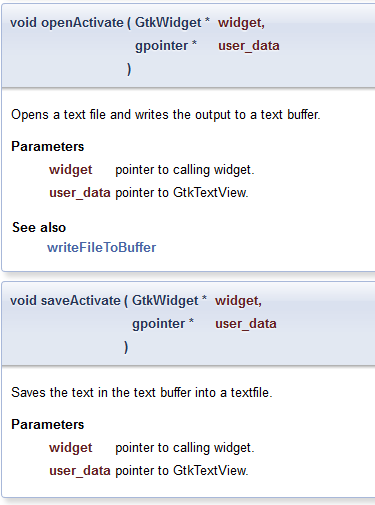
## Assembler GUI docs

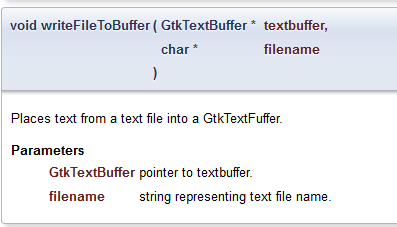




## 



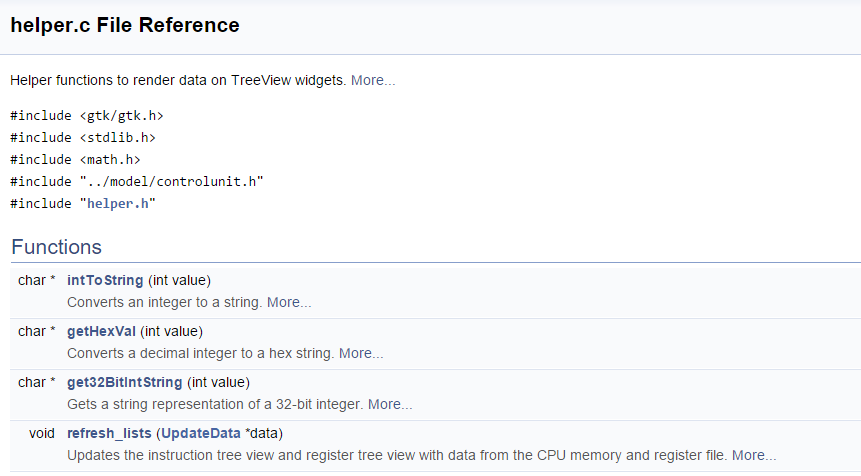




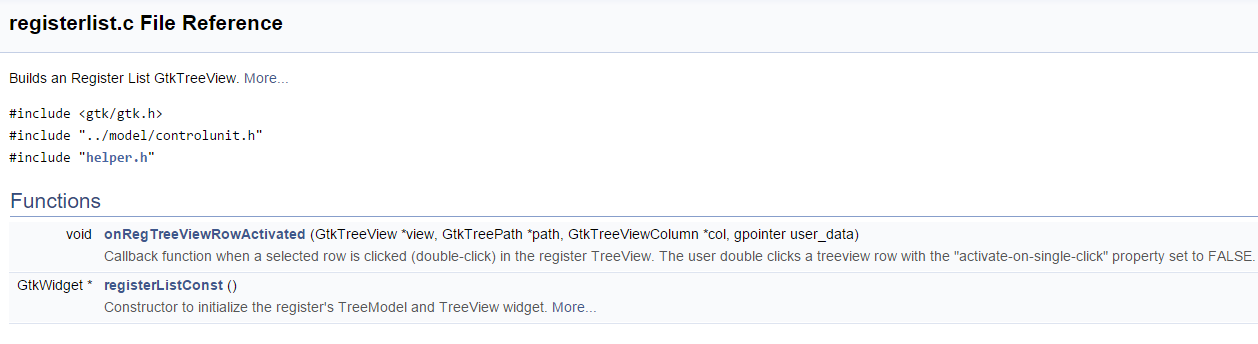
# Appendix C

## Simulator Main docs

**Helper docs**

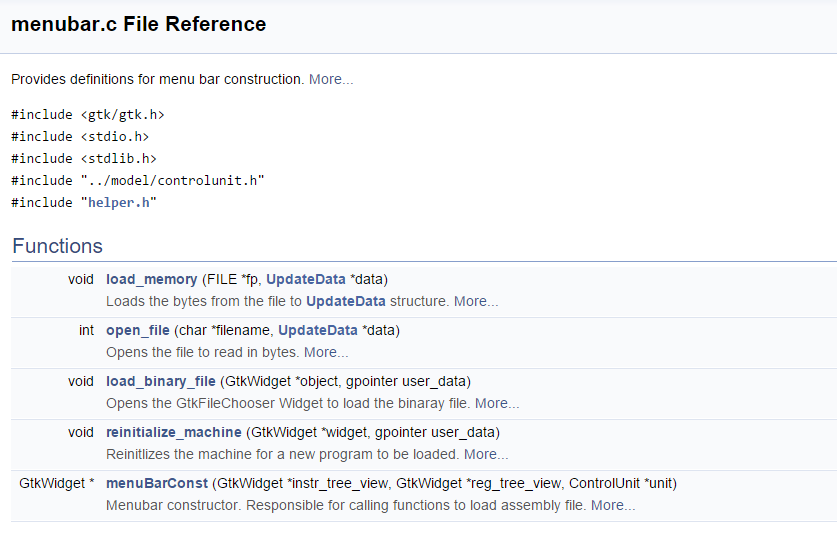


## List docs



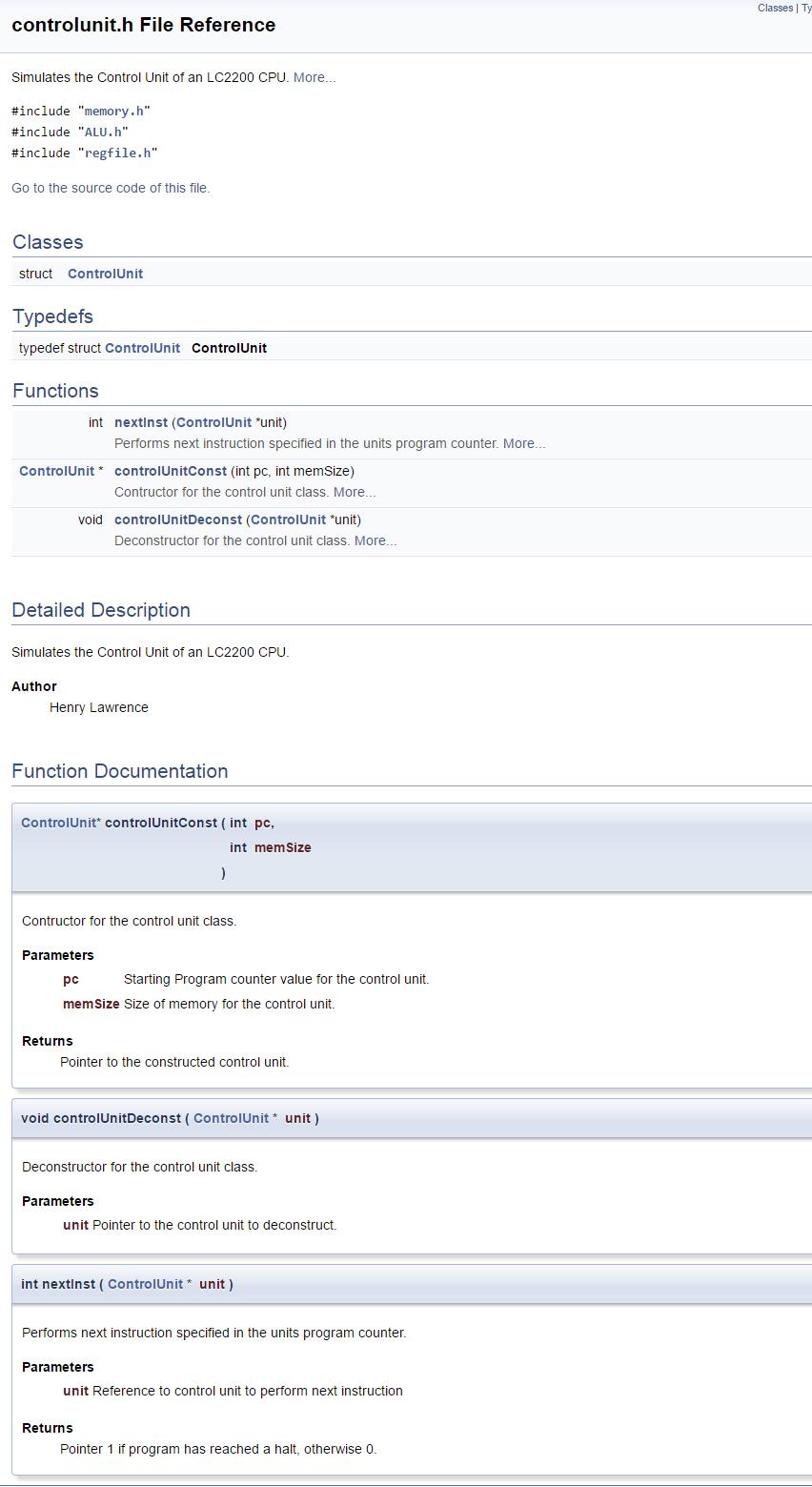


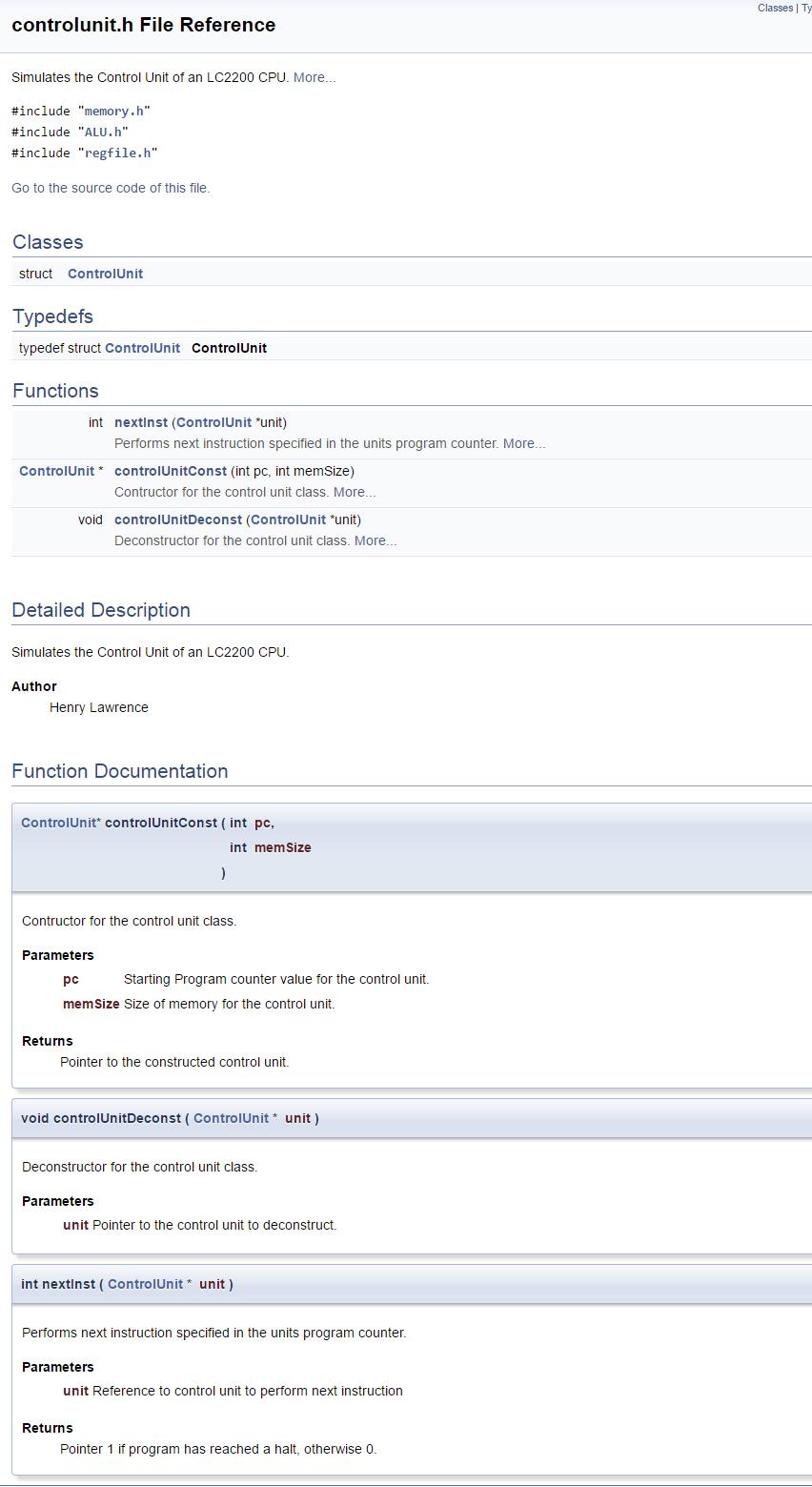
## MenuBar docs



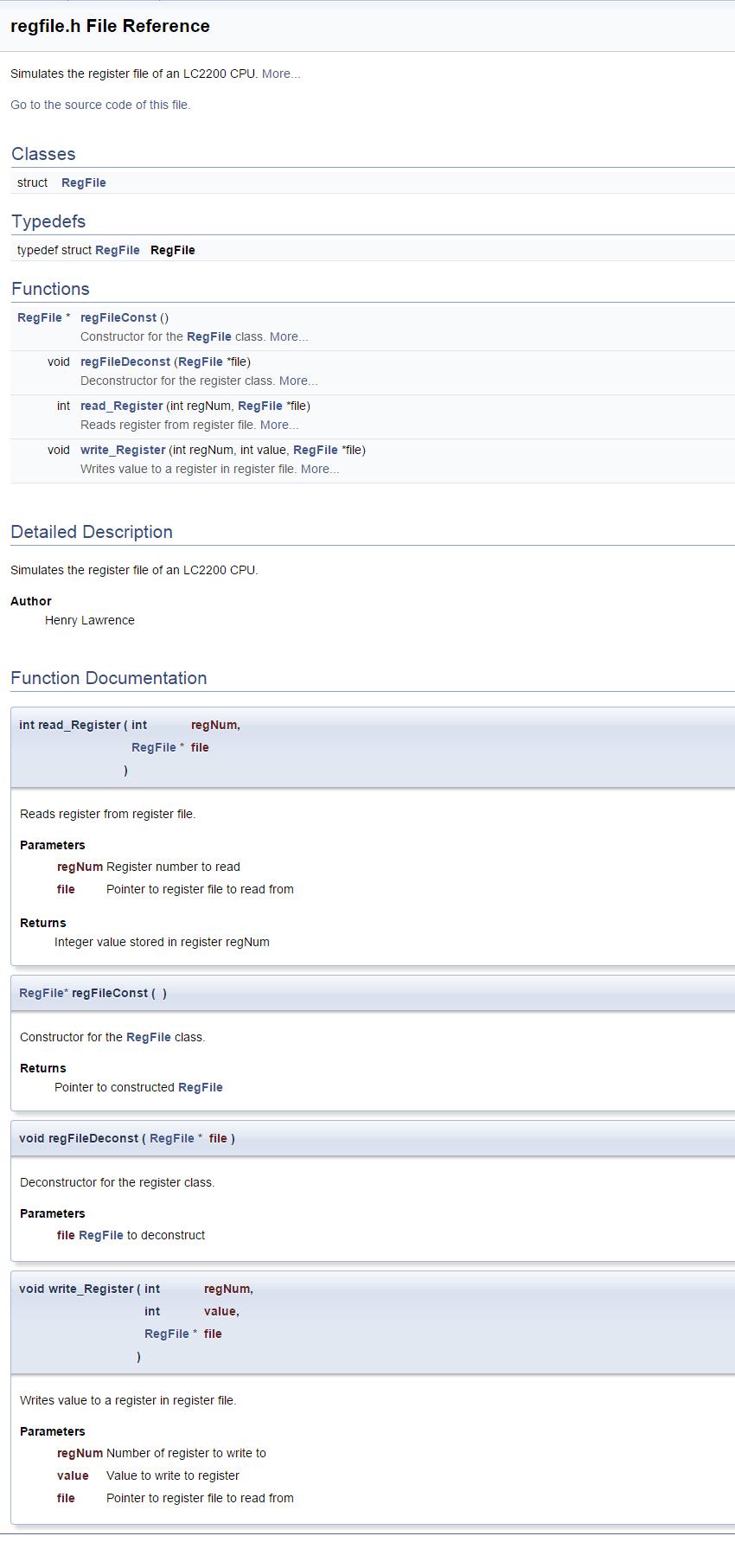
# Appendix D

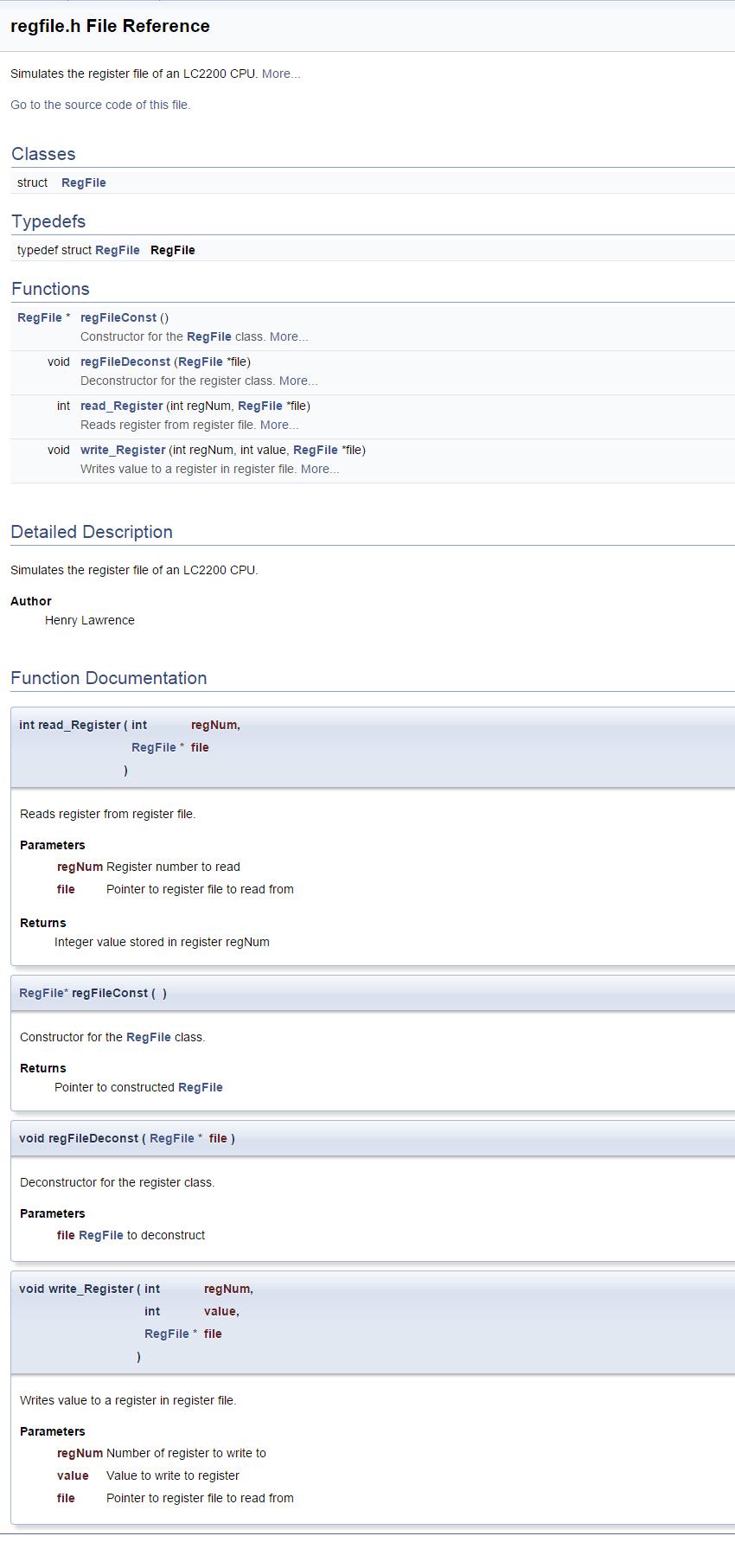
## Control Unit docs



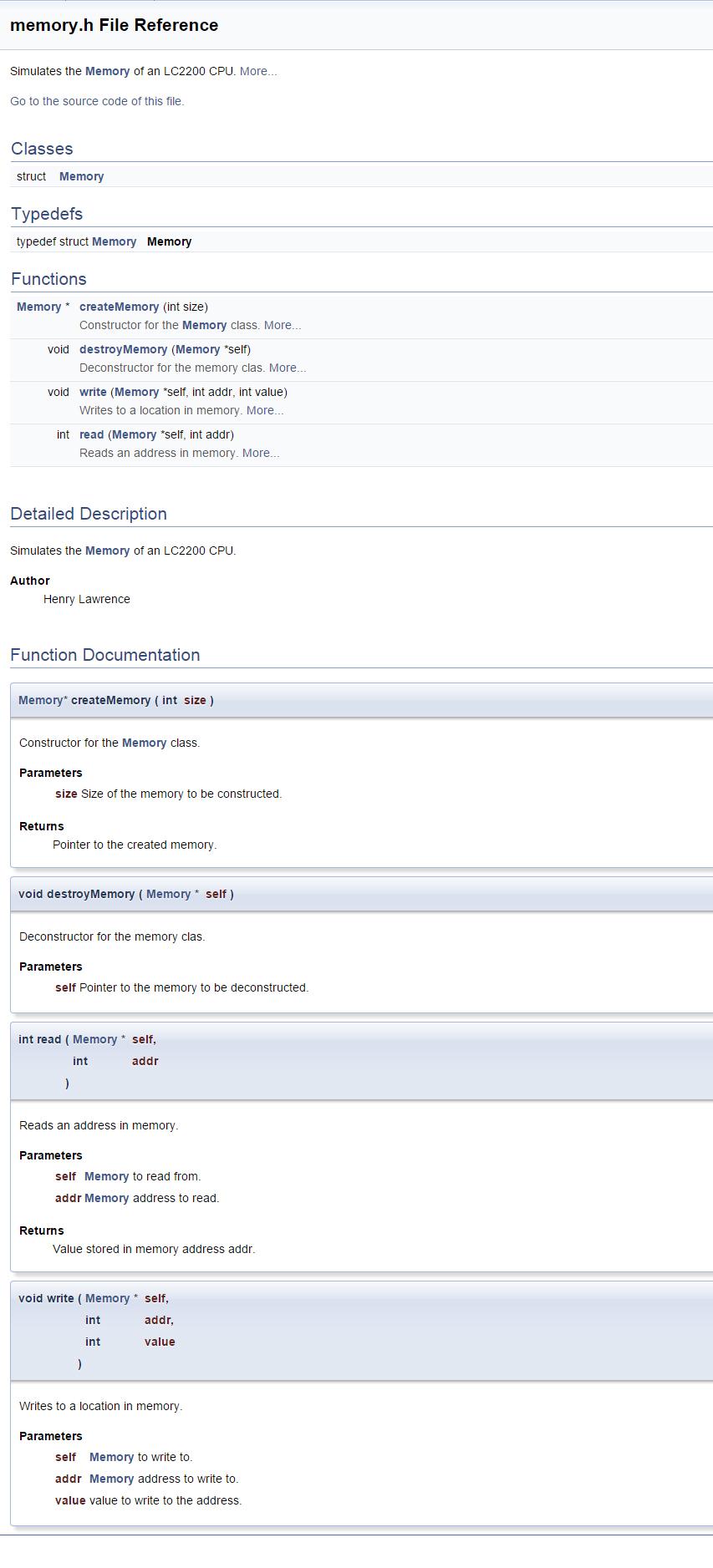


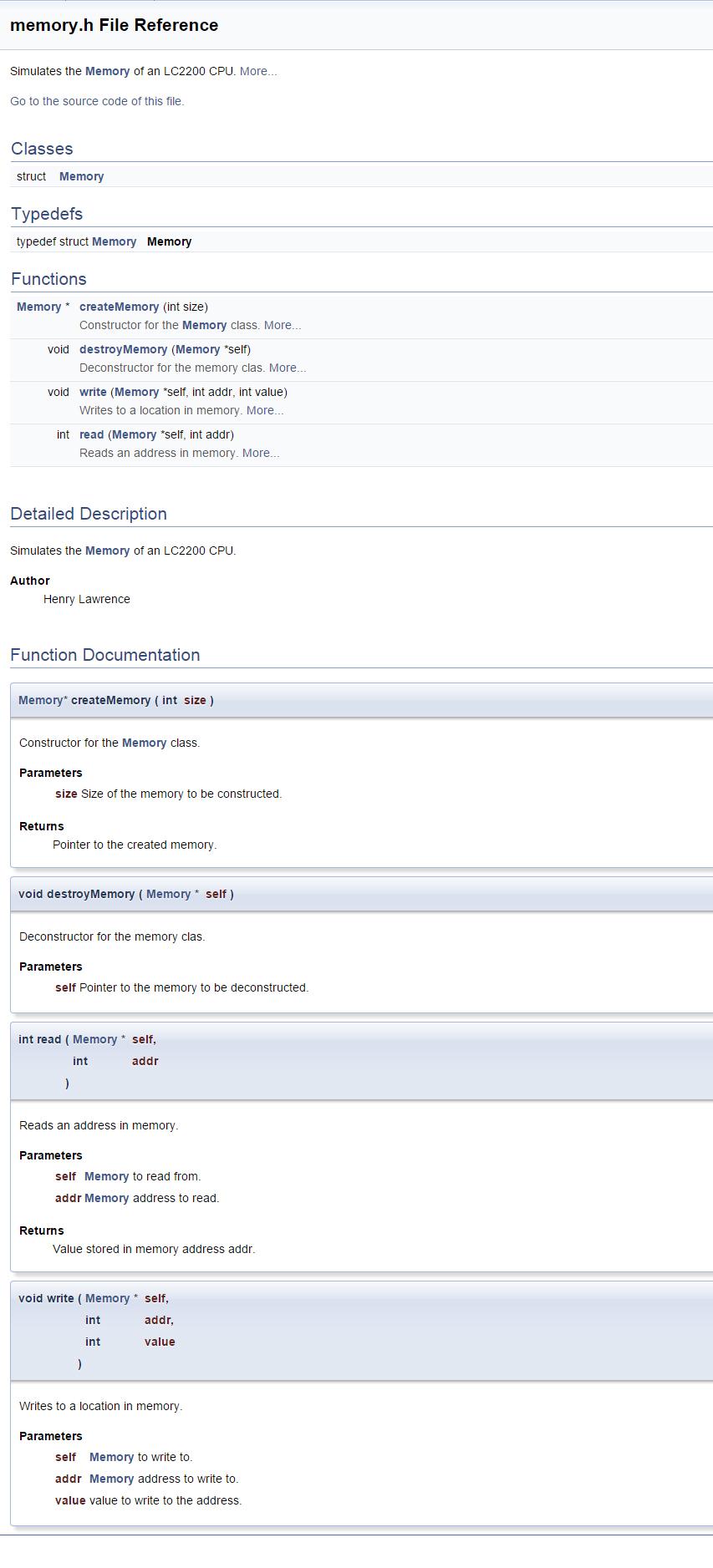
## Register File docs



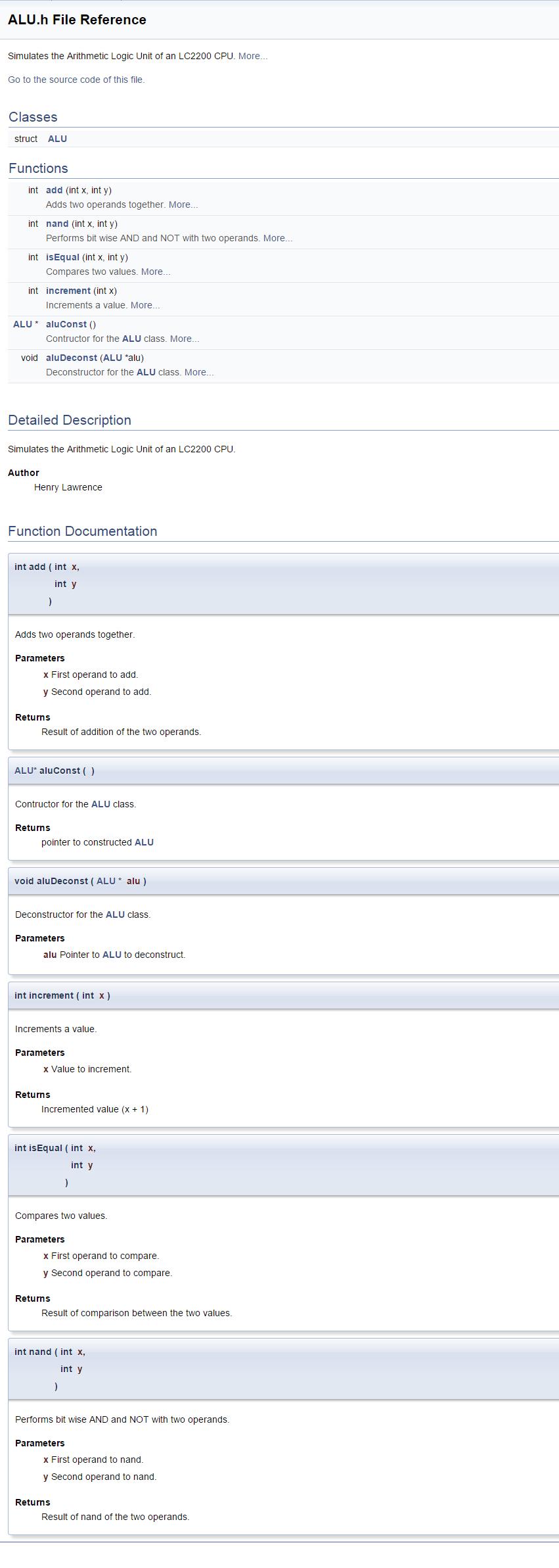


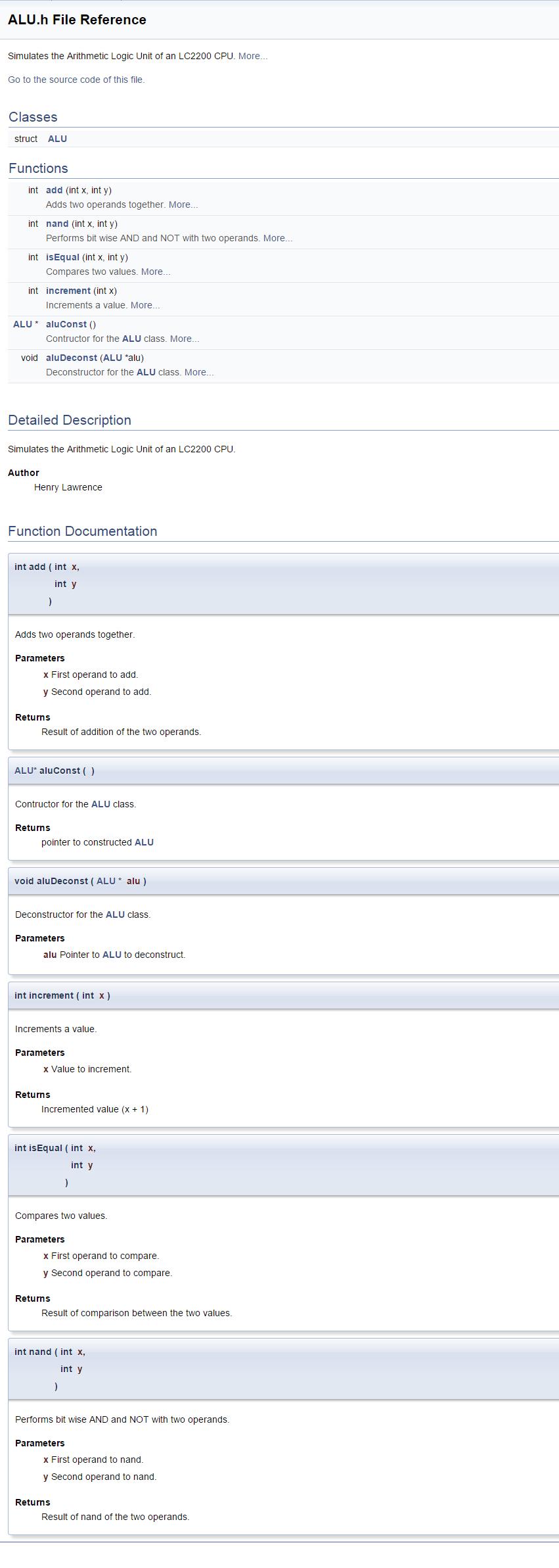
## Memory docs

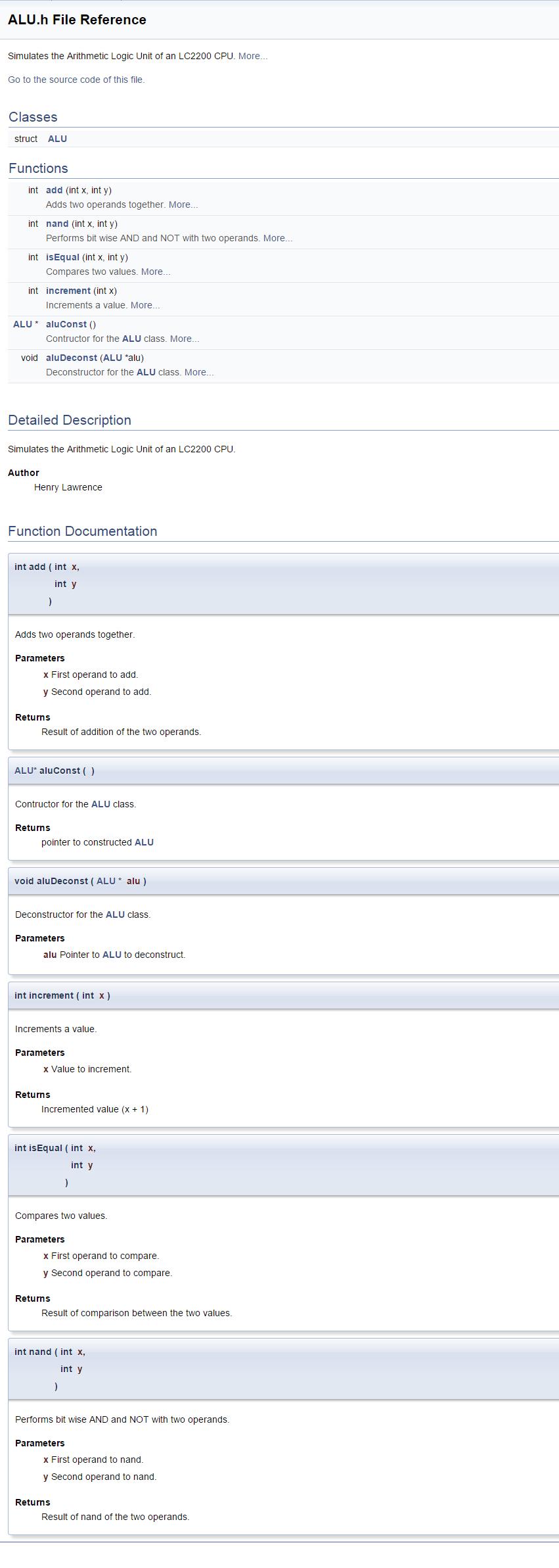




## ALU docs







## LC-2200 UML

