UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

Architecture and Organization of Computers

Phase One Report

Presented by:

Samuel Matos	samuel.matos@upr.edu
Johanna Rivera	johanna.rivera2@upr.edu
Victor J Reventos	victor.reventos@upr.edu

For: Nayda G. Santiago
Course: ICOM 4215, Section 020
Date: April 17, 2013

Abstract

This report contains all the criteria used to design and develop a simulator for the simple processor, RISC AR5. It describes the outlying structure, main components, specifications and design strategies used when developing the RISC R5 processor. The simulator is designed to run on a Java® Virtual Machine for cross-platform support, providing users with a visual representation of key behavioral aspects included in most RISC machines nowadays.

Table of Contents

Introduction

Modern architectural advancement has contributed heavily to the evolution of computers. Early processors were units capable of performing a limited set of arithmetic calculations such as addition and subtraction. Nowadays, processors serve a much bigger role when compared to their original purpose; though the fundamental operation hasn't change. Most processors allow the execution instructions that perform arithmetic, logical and input/output operations within many different computer systems.

The main characteristics that distinguish a processors prowess are the instruction set, its bandwidth and clock speed. The instruction set encompasses the different instructions the processor can execute whereas the bandwidth determines the number of bits that can be processed per instruction. The frequency factor known as the clock speed defines the amount of instructions per second that can be executed in processors.

Processors are normally categorized into CISC or RISC paradigms. The Reduced Instruction Set Computing (RISC) focuses primarily on providing a simpler instruction set, tightly couple with hardware limitations in order to provide higher performance and faster execution cycles. Complex Instruction Set Computing (CISC) in contrast with RISC, focuses on providing a more complex instruction set, ultimately limiting hardware potential.

Specifications

This project consists of delivering a fully functional processor simulator based on the RISC AR5 design. The RISC AR5 processor is designed by Nayda G. Santiago, taking the ideas from the Simple Risc Processor, from the Jordan and Heuring textbook, a processor designed by Manuel Jimenez, Sunil Vaidya, Bradley Vansant, and Dave Dorner for the EE 813 course at Michigan State University, and the processor designed by Adem Kader and Mustafa Paksoy for the E25: COMPUTER ARCHITECTURE course at Swathmore University.

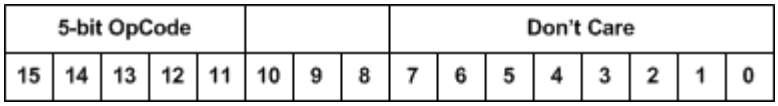
The RISC AR5 features:

- One Internal 256-word 8-bit wide program memory
- One 8-bit accumulator
- One 4-bit status register
- One 8-bit program counter
- One 16-bit Instruction register
- Eight 8-bit general purpose registers
- Two external I/O pins
- One 8-bit internal data bus
- One 4-bit hardware multiplier
- Instruction set of 20 instructions
    - Five arithmetic
    - Four logical
    - Five data transfer
    - Four control flow instructions
    - Two machine control

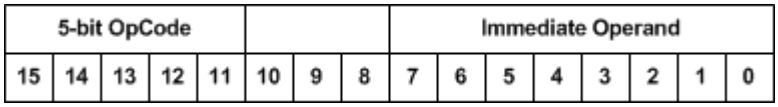
The RISC R5 Memory consists of 256 addresses, each address containing an 8-bit cell. The memory's Endianness consists of big-endian ordering, meaning that the most significant byte is stored first or read first. Instructions are loaded into the main memory from address 0 to 127 and I/O ports are connected using the addresses 250-251 for input and 252-255 for output.

The processor has eight 8-bits general purpose registers named from R0 to R7, including, four special purpose registers known as the Program Counter (PC), Accumulator (A), Status Register (SR) and Instruction Register (IR). The Program Counter and Accumulator have a common size of 8-bits whereas the Instruction Register possesses a size of 16-bits and the status register a size of 4-bits, each bit representing the value of a different flags known as the Zero Flag (Z), Carry Flag (C), Negative Flag (N) and Overflow Flag (O). Negative numbers are represented using two's complement and R7 is a register meant to be used for branching locations.
The supported addressing modes are: Implicit, Immediate, Direct and Register Direct. Their distinctive functionality is as follows:

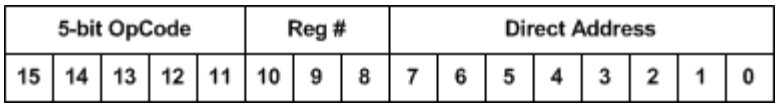
- Implicit – The only operand needed is contained in the accumulator.



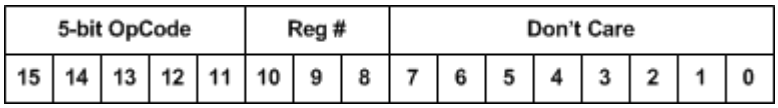
- Immediate – The data to be operated is part of the instruction.



- Direct – The memory location is indicated in the instruction.



- Register indirect – The register contains the operand.



The instruction set of the RISC AR5 is summarized in the following table:

Item	Opcode	Name (Mnemonic)	Operand	Addressing Mode	Operation	Details
1	00 000	AND rf	Accumulator, register f	Register Direct	A ← A and rf	Logical AND
2	00 001	OR rf	Accumulator, register f	Register Direct	A ← A or rf	Logical OR
3	00 011	ADDC rf	Accumulator, register f	Register Direct	A ← A + (rf) + carry	Addition with carry
4	00 100	SUB rf	Accumulator, register f	Register Direct	A ← A - (rf)	Subtraction
5	00 101	MUL rf	Four least significant bits of accumulator, four least significant bits of register f	Register Direct	A ← A * (rf)	Multiply
6	00 110	NEG	Accumulator	Implicit	A ← ^(A)	Two's complement
7	00 111	NOT	Accumulator	Implicit	A ← not(A)	Negate
8	01 000	RLC	Accumulator	Implicit	A ← A6…A0 & CF, CF ← A7	Rotate left through carry
9	01 001	RRC	Accumulator	Implicit	A ← CF & A7…A1, CF ← A0	Rotate right through carry
10	01 010	LDA rf	Accumulator, register f	Register Direct	A ← (rf)	Load accumulator from register f
11	01 011	STA rf	Accumulator, register f	Register Direct	(rf) ← A	Store accumulator to register f
12	01 100	LDA addr	Accumulator	Direct	A ← [addr]	Load accumulator from memory location addr
13	01 101	STA addr	Accumulator	Direct	[addr] ← A	Store accumulator to memory location addr
14	01 110	LDI Immediate	Accumulator	Immediate	A ← Immediate	Load accumulator with immediate

15	10 000	BRZ	Status register	Implicit	If Z = 1, PC ← R7	Branch if zero
16	10 001	BRC	Status register	Implicit	If C = 1, PC ← R7	Branch if carry
17	10 010	BRN	Status register	Implicit	If N = 1, PC ← R7	Branch if negative
18	10 011	BRO	Status register	Implicit	If O = 1, PC ← R7	Branch if overflow
19	11 111	STOP	Program counter	Implicit		Stop execution
20	11 000	NOP		Implicit		No operation

The simulator is able to provide a visual representation of the RISC AR5 instruction set, including the behavior of different addressing modes. When Instructions are loaded into the simulator using memory ranges 0 to 127, the Program Counter resets to zero and starts increasing as the user steps through the instructions. The supported methods for execution are:

- Run: Executes all remaining instructions.
- Step: Steps through each instruction, showing each change to the processors components.

The RISC AR5 simulator shows the contents of all general purpose registers, special purpose registers and memory as the user steps through each instruction. Memory values are displayed in hexadecimal representation for easy debugging of memory changes. Also, a file display is included, showing the contents of the file that contains the instructions to be loaded into the system, marking any errors that might appear if invalid instructions are used. The following guideline is used in order to comply with a valid file:

- Instructions represented in four hexadecimal digits.
- One line per instruction,
- Valid OpCodes.
- File has no more than 64 lines.
- Last instruction detected is the stop instruction.

Last but not least, the simulator tries to depict the behavior of I/O ports. Inputs are loaded using the user's keyboard; typing in the provided input box makes the values available at addresses 250-251. Output is available as a four digit ASCII display that shows the contents of memory addresses 252-255 at all times.

Design

Use Case

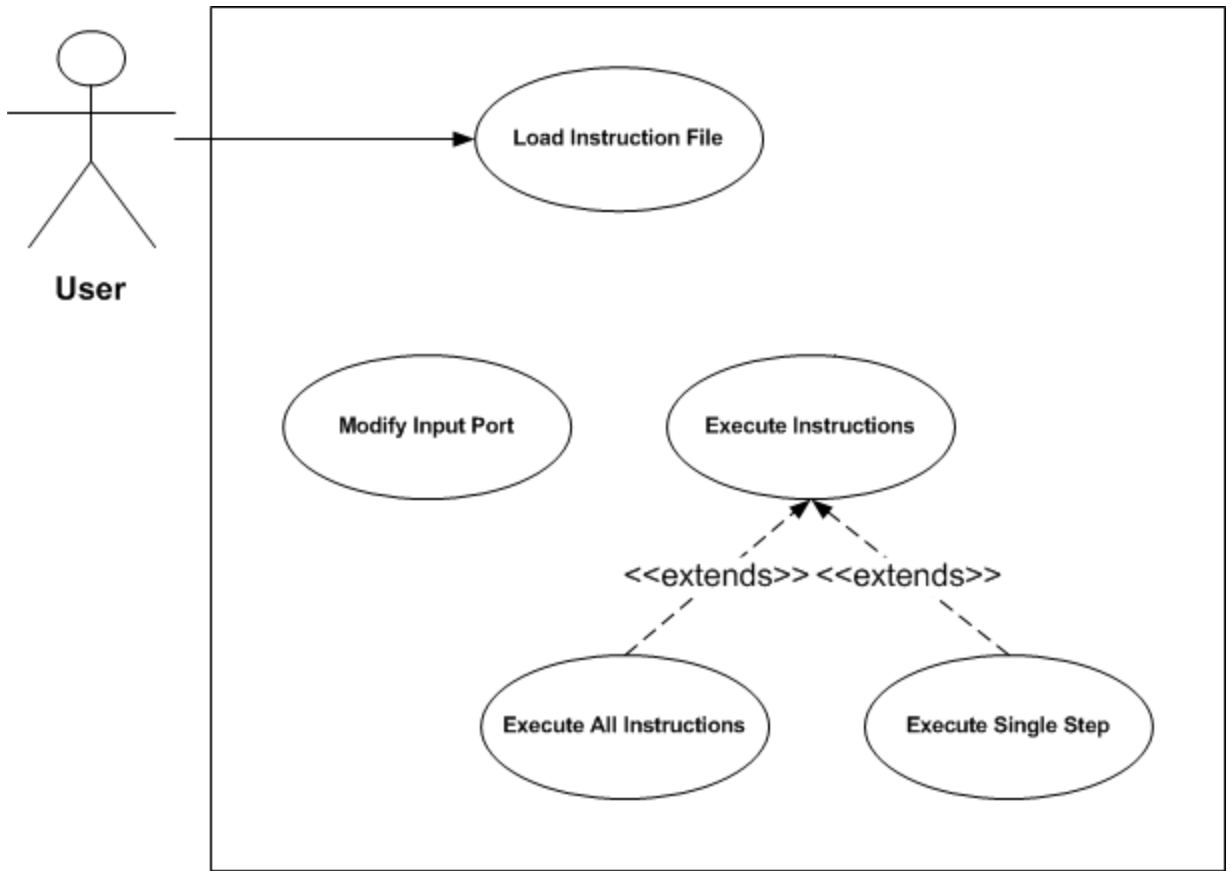


Figure 1 RISC AR5 Simulator Use Case Diagram

The above use case diagram depicts the actions the user can do while running the RISC AR5 simulator. The user starts the interaction with the simulator by loading a valid file containing the instructions that will be executed by the simulated processor. After loading the file, the user has two options that he can always choose from; either modify the input port or execute instructions in step by step or run mode.

The simulator was designed assuming that the data is processed in two's complement representation. After analyzing the requirements we came up with a solution which consisted of both Object Oriented and Procedural style programming. Most of the processor's implemented algorithm follows an Object Oriented design whereas some components such as the Arithmetic Logic Unit, including its Flag Management were divided into procedures. The main interfaces for the simulator are:

Module	Description
UI	This module depicts all the graphic aspects used by this simulator. It acts as a view and a controller. It mainly controls the processor, the File loader and the IO Channel which is used for I/O between the user and processor. It also controls all the needed modules to achieve instruction execution in either step or run mode.
Processor	Executes the provided instructions. It works in two modes; step and run. The run mode executes all the remaining instructions.
File Loader	Loads the instructions from a file and validates the file being loaded.
Memory	This interface provides communication for a memory module. It provides methods for getting and setting the data into memory. The data being stored is masked to the cell size of the RISC R5 memory module.
Register	Provides a way to specify the Register width. By using the width it determines if the data being provided fits in the specified number of bits. Also, it provides setters and getters for the data being stored in the register.
Status Register	Special interface providing the same operations as a Register, with included support for flag operations such as getters, setters, and clear all.
Instruction Set	The instruction set is divided into different instruction categories: arithmetic and logical, load/store and program flow. The instruction set interface provides all the functionality supported by this processor.
Flag Management	This module uses the operands and the result in order to make manage flags contained within the status register.
I/O Channel	Interface that provides communication with the input and output ports. It is initialized to the characters that can be used with the ports.

Functional Block Diagram

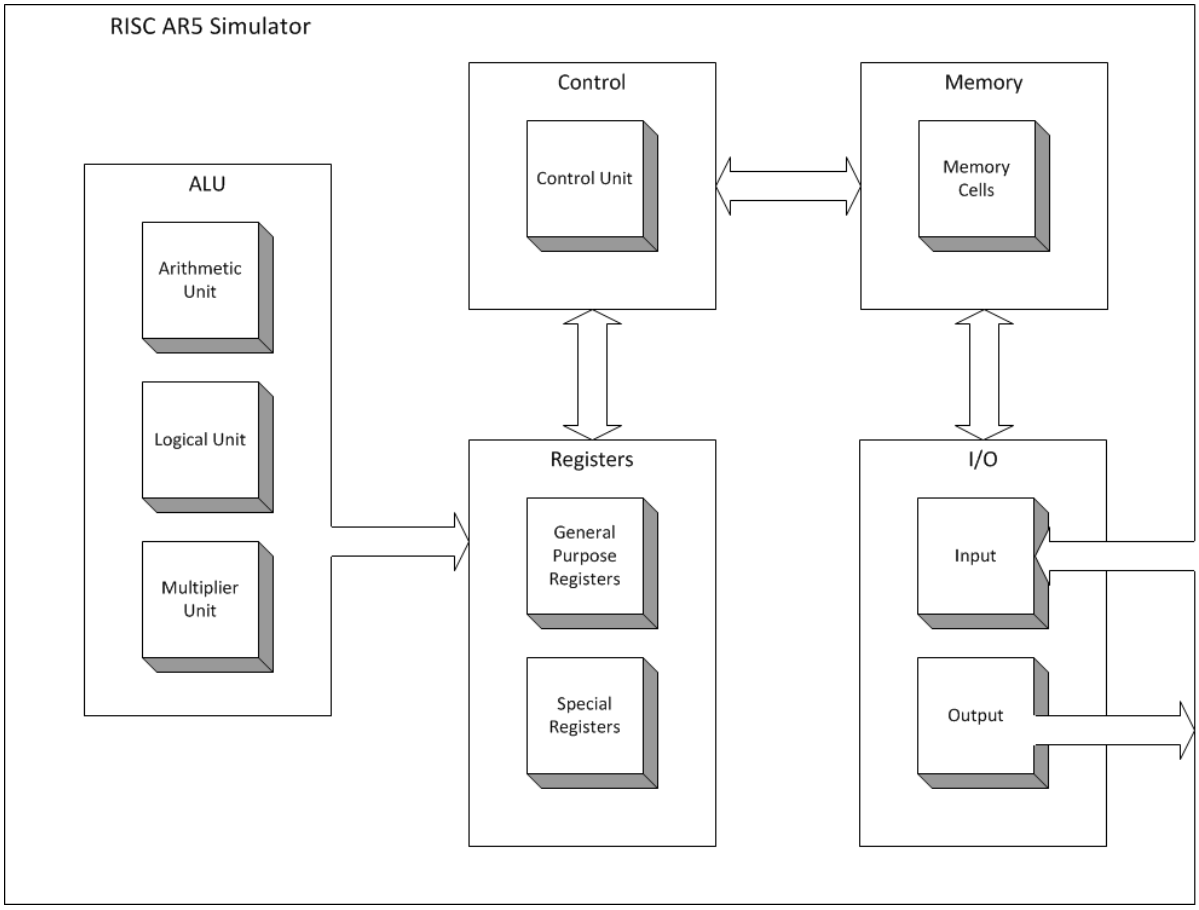


Figure 2 RISC AR5 Block Diagram

The RISC AR5 simulator behaves exactly as a processor would normally do. In the block diagram above we can see the different components belonging to a processor. The ALU performs arithmetical and logical operations that interact directly with the accumulator and other registers in order to load or store results in an immediate fashion. The control unit makes sure that the communication in the entire system stays synchronized; assuring instructions are fetched, decoded and executed successfully. The memory consists of the program memory and data memory, storing all instructions that will get executed and all data that wants to be stored or loaded from memory. Finally, the I/O ports serves as a mean to interact with the processor from the exterior, allowing the interconnection of multiple components.

Class Diagram

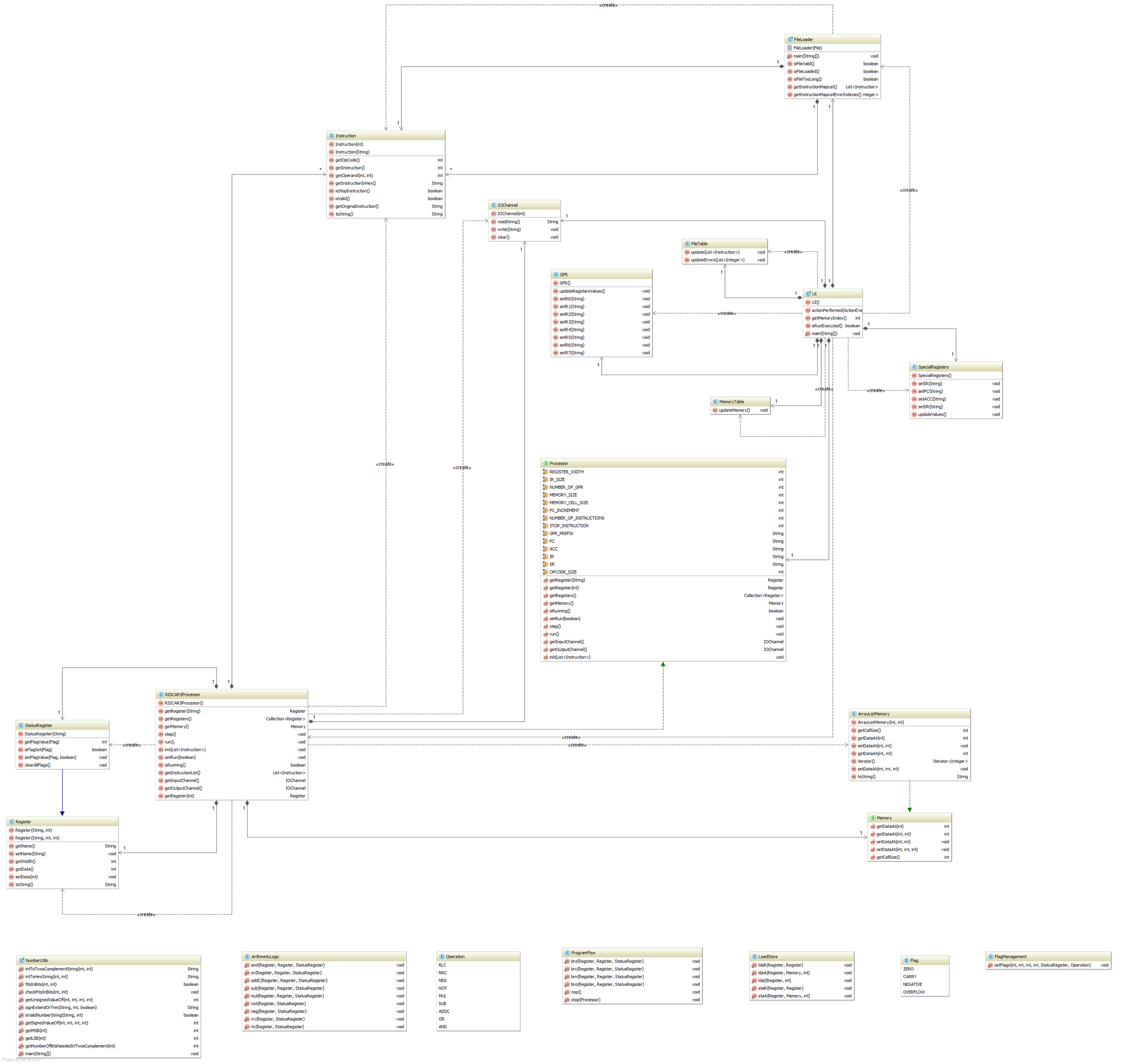


Figure 3 Phase One Source Code Class Diagram

The class diagram above depicts the underlying structure of our high level design of the RISC AR 5 processor. The back-end supporting the user interface and acting as the control unit is the RISCAR5Processor class which implements the Processor interface. This class interacts directly with the memory, the registers and the Arithmetic and Logic Unit in order to provide fetching of instructions, decoding and execution of instructions. The front-end is encompassed by the UI class, possessing the capability of communicating directly with the processor, allowing the reading all the registers contents, memory contents and I/O port status.

Work Breakdown Structure

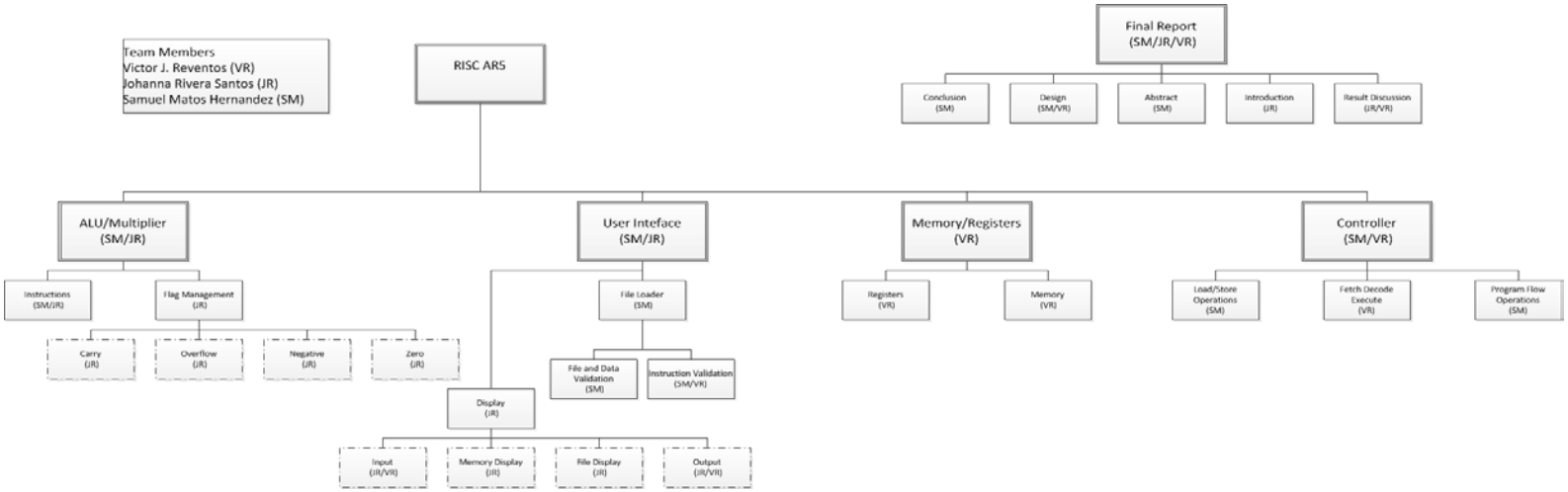


Figure 4 Phase One Work Breakdown Structure

The work breakdown structure consists of two main modules; the RISC AR5 simulator and the final report. The simulator is divided into the following sub-modules: ALU/Multiplier, Memory/Registers, Controller and User Interface. The corresponding tasks for the ALU/Multiplier sub-module are Instructions and Flag Management, which were distributed between Samuel Matos and Johanna Rivera. The implementation of Memory/Registers sub-module corresponded solely to Victor Reventos. The Controller contains Load/Store and program flow operations, including the Fetch, Decode, and Execute cycles. The tasks of the Controller sub-module were delegated to Samuel Matos and Victor Reventos. The User Interface is divided into the graphical user interface and the File Loader, which also includes validation of instructions and data; these were distributed between all three members. Most of the tasks were balanced by the workload of each person, taking into consideration the different task each one had in respect to other courses. The distribution of task was balanced to the greatest extent, in order to allow everyone the benefit of learning how everything interconnects.

Results

Screenshots

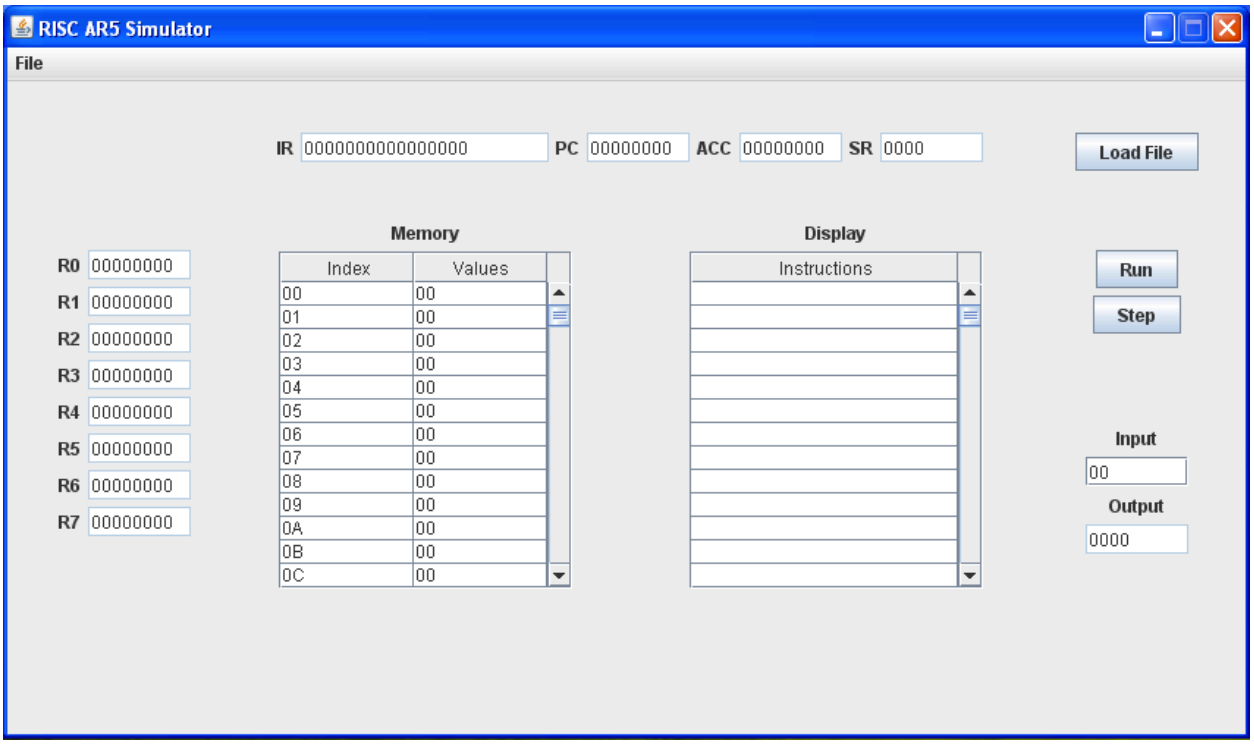


Figure 5 The RISC AR5 Simulator

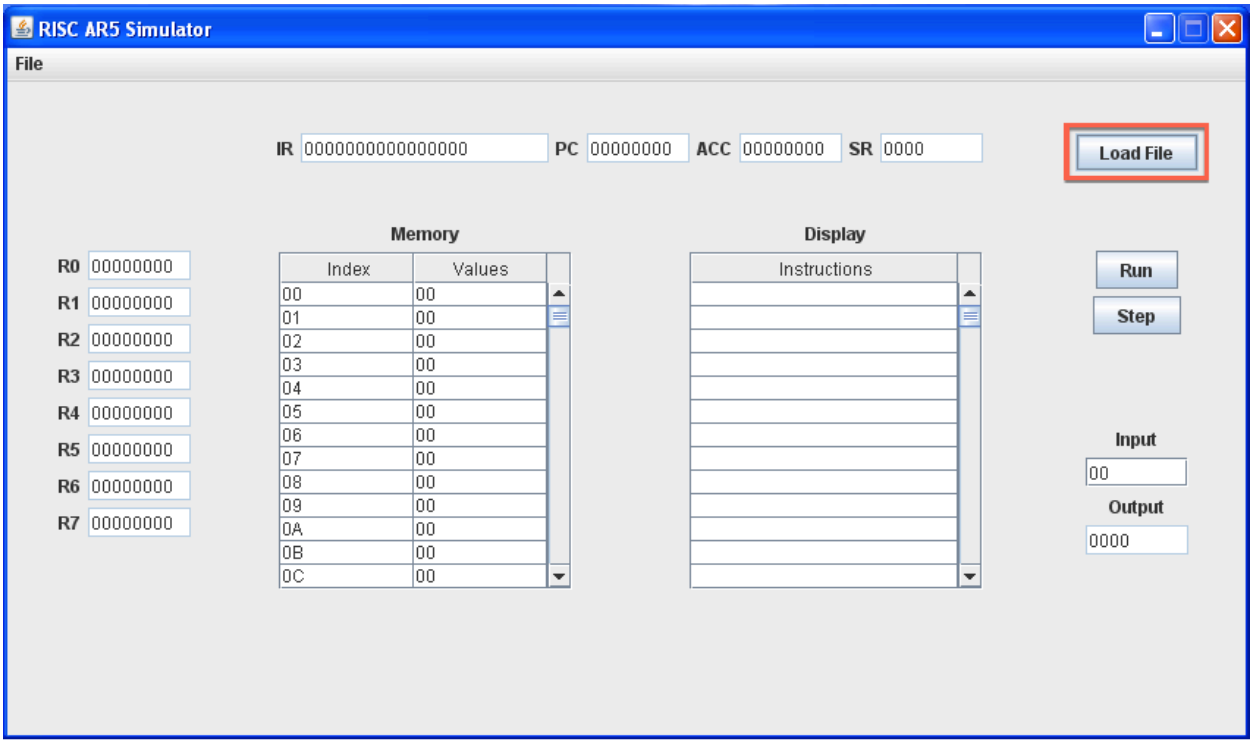


Figure 6 This image sample shows how to load a file to the simulator.

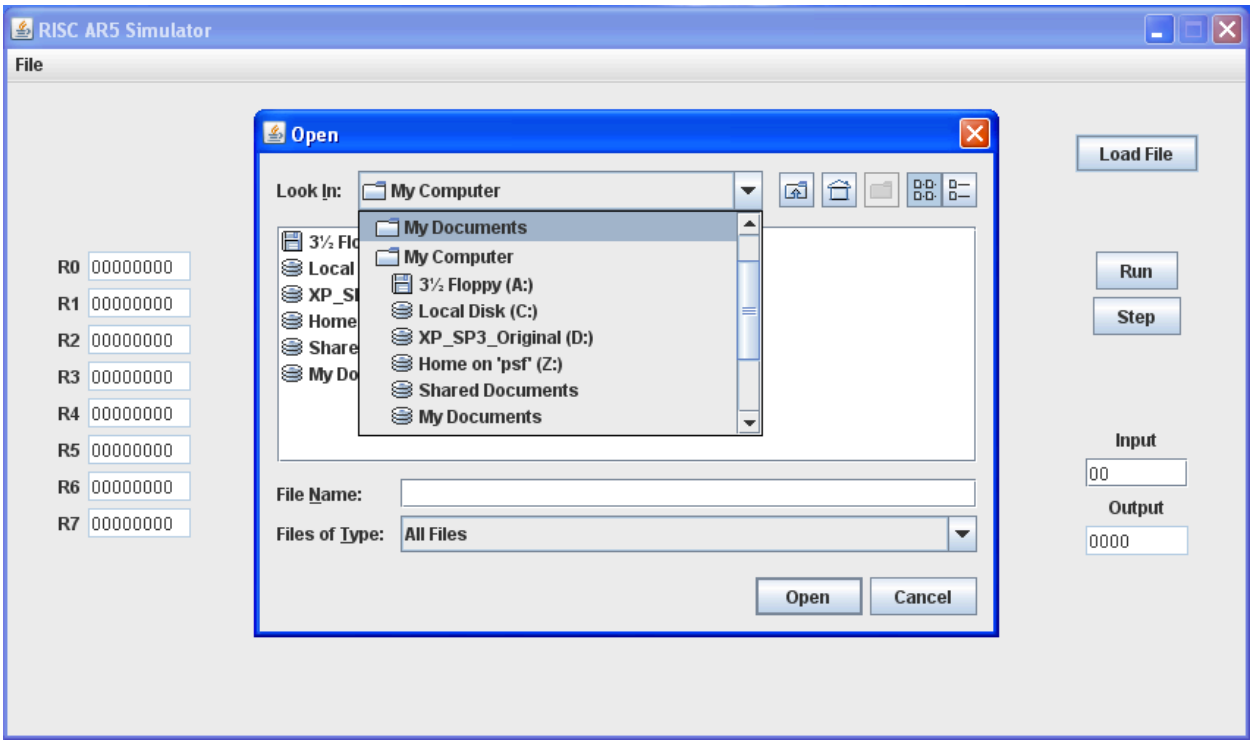


Figure 7 After pushing "Load File" button, the file browser appears on screen.

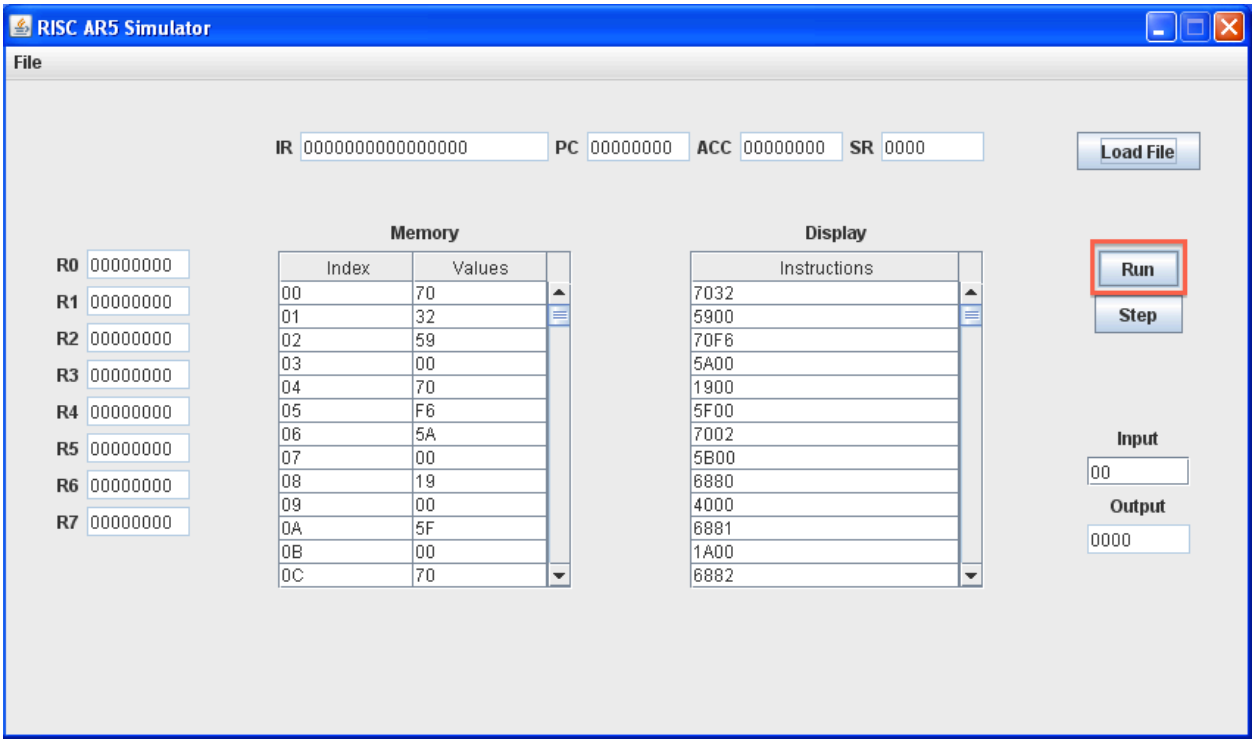


Figure 8 The file is successfully loaded to memory and ready for execution.

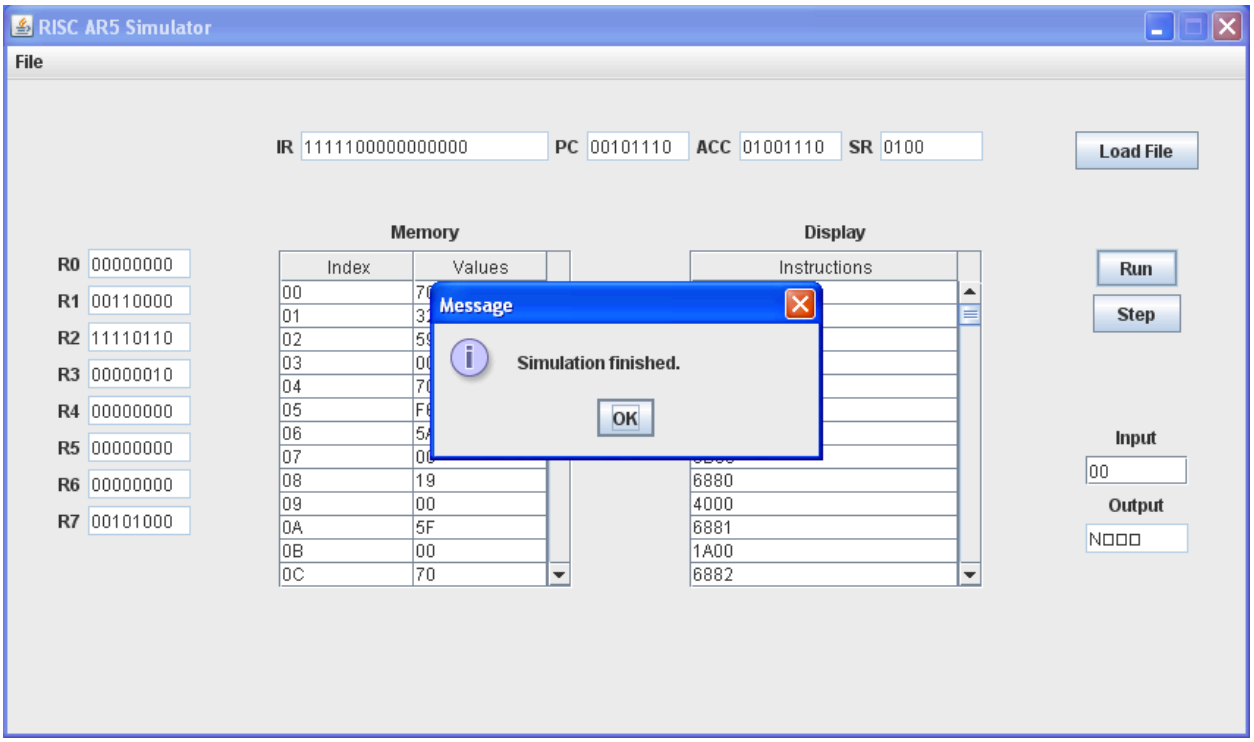


Figure 9 After the "Run" button is pressed the simulation is executed.
The message "Simulation finished." appears on screen.

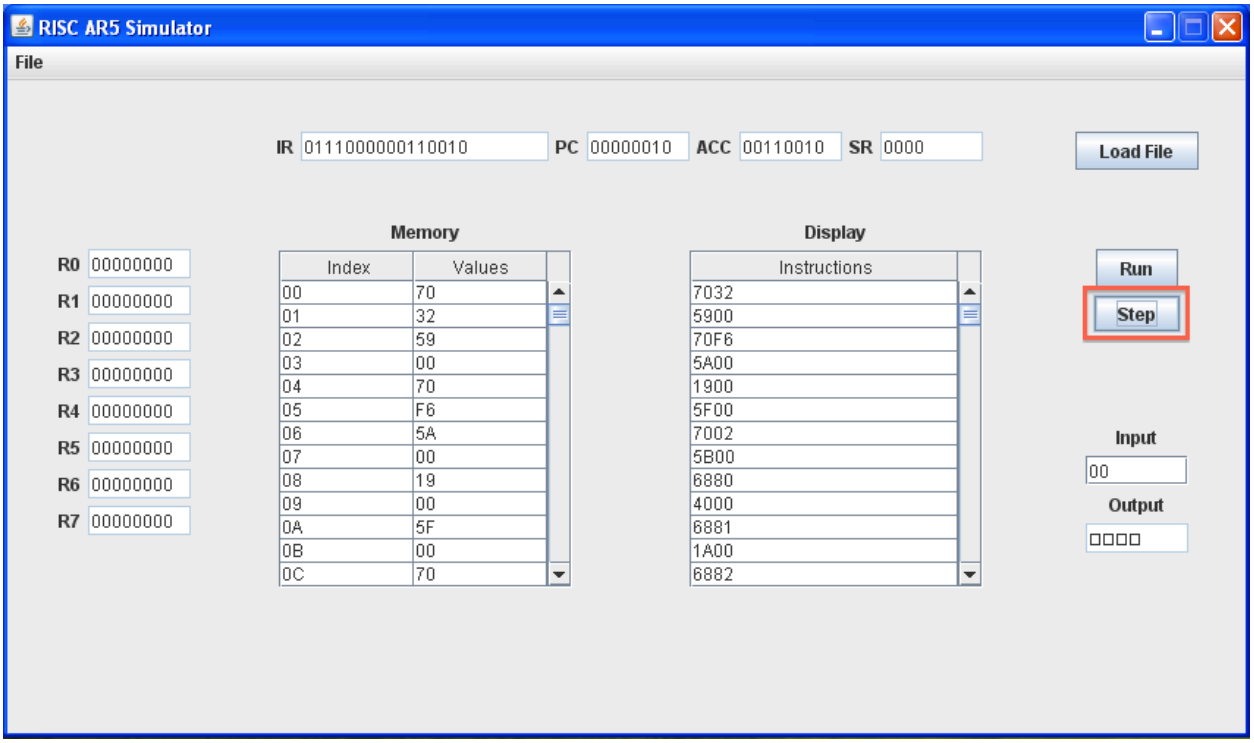


Figure 10 The "Step" button executes one instruction at a time.

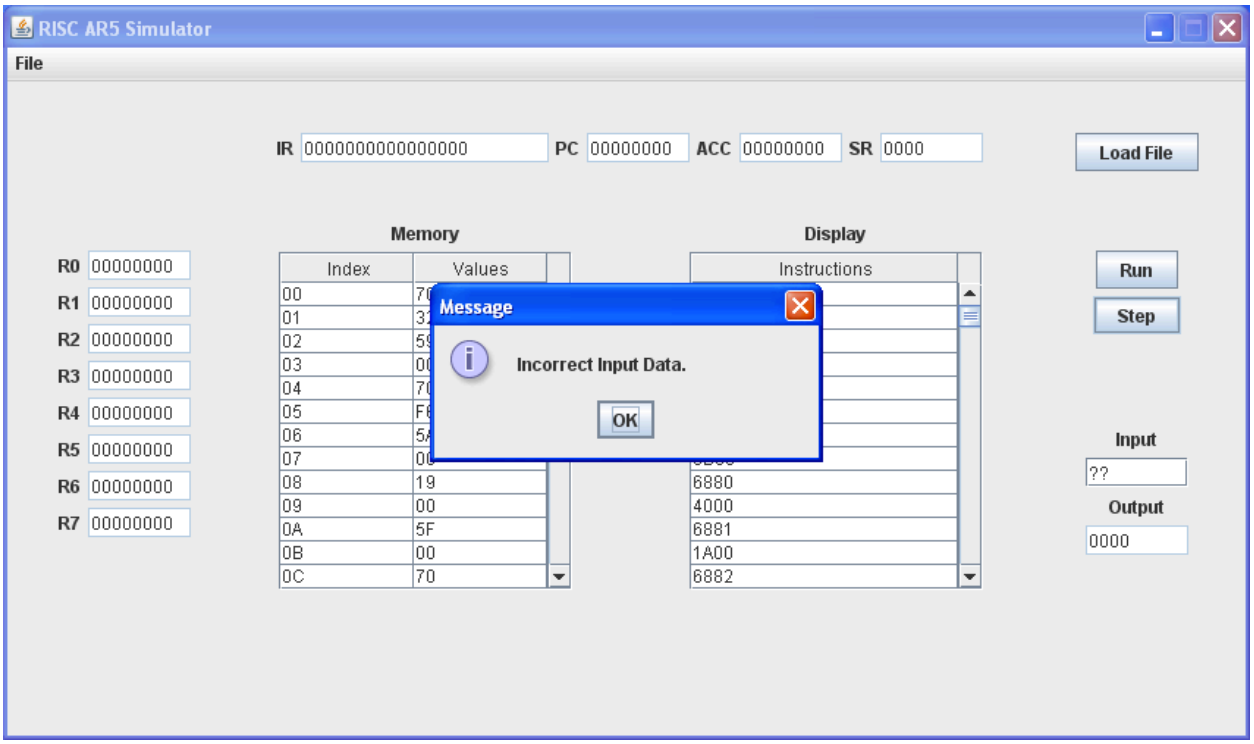


Figure 11 The message "Incorrect Input Data" appears when incorrect input is entered.

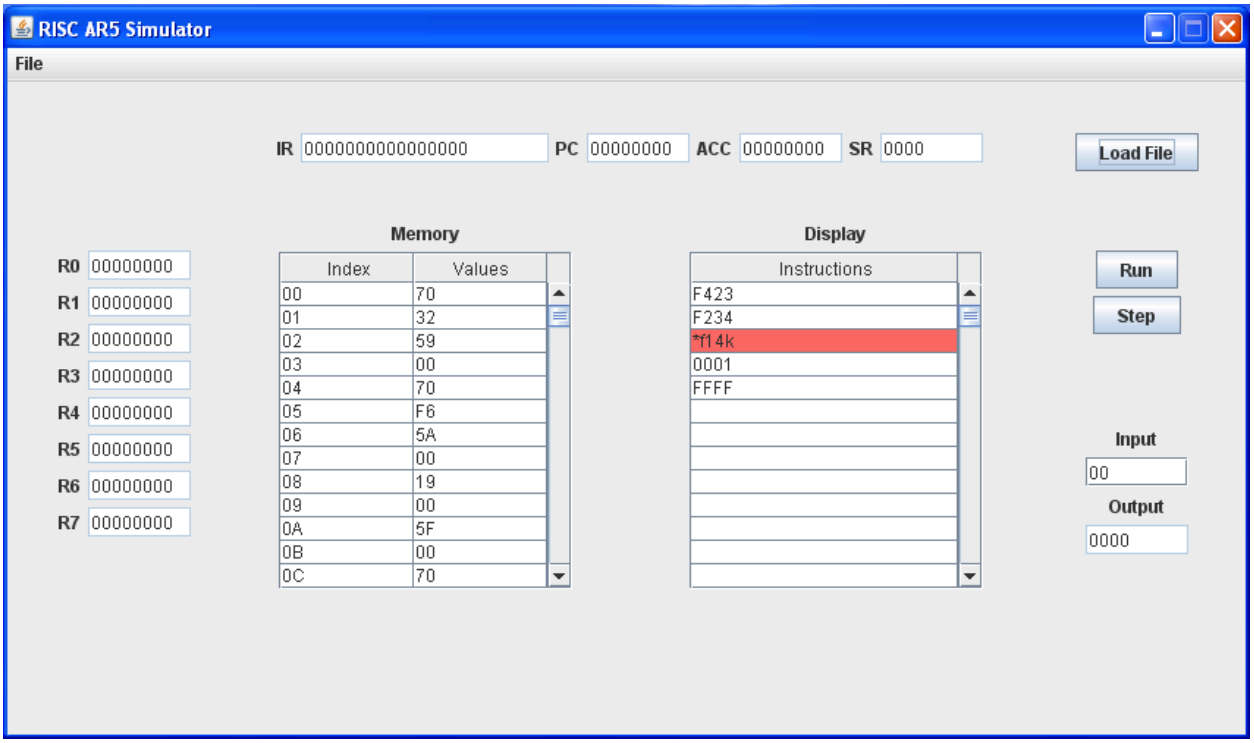


Figure 12 The simulator marks the invalid instructions of a loaded file with the color Red.

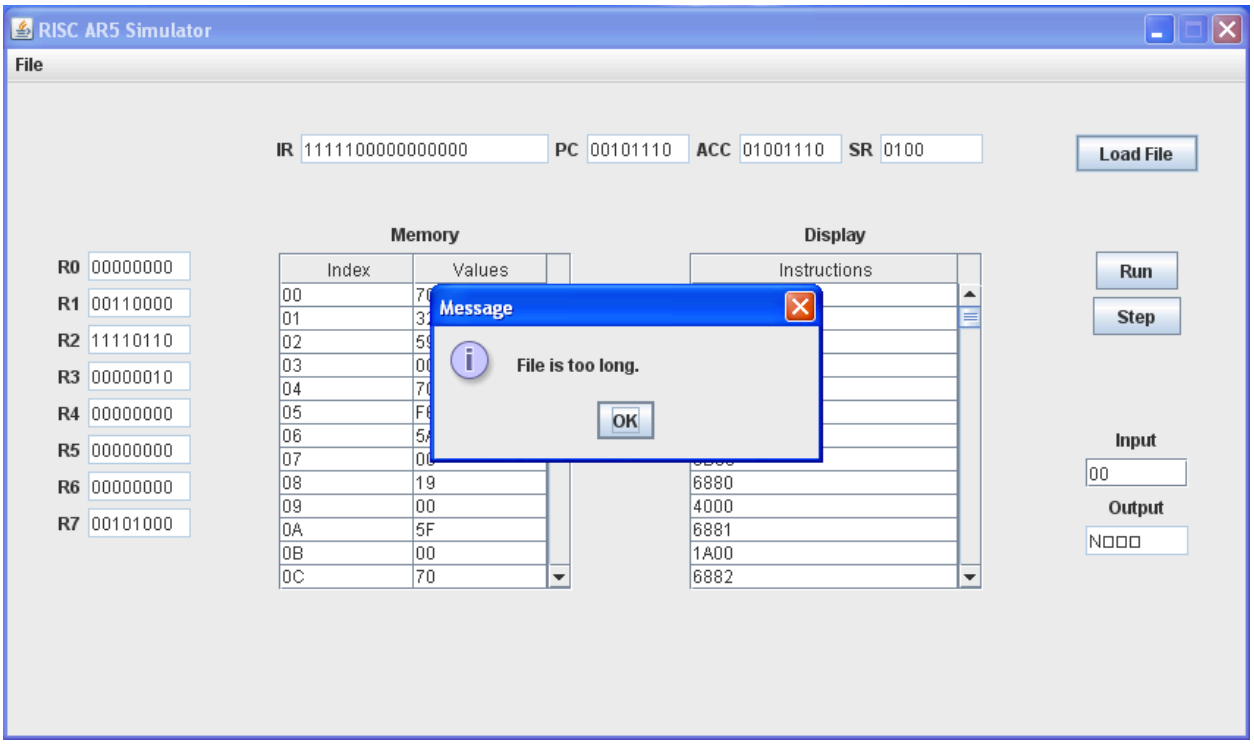


Figure 13 The message "File is too long." appears on screen when the number of instructions in the loaded file is bigger than the limit.

Testing

For testing purposes we used the Java de facto standard known as JUnit. Testing was performed using bottom-up approach, which is, creating test for each individual module, testing them, and merging them in a hierarchical fashion. Each test follows the black-box methodology, separating expected results from the implementation details, letting us know if the system is behaving as it should be. External and internal boundaries were also tested in order to provide a robust simulator to the user.

The system was tested in different testing levels such as unit, integration and system testing. After testing each module individually, integration tests were prepared to verify if the processor was behaving as it should when combined with all of its components. Finally, system testing allowed us to know if the system was fully integrated and functional. We believe that this level of testing is required in order to verify the complete functionality while performing assessment on its robustness.

Source Code

The source code for the simulator is attached to this report.

Conclusion

The successful implementation of the RISC AR5 simulator allowed us to grasp the key concepts involved when designing and implementing a processor using a certain degree of abstraction. Each team member was exposed to critical thinking in their respective tasks due to the nature of the work, allowing us to apply knowledge gained throughout our academic years while incorporating new aspects learned about the architecture and organization of computers.

The final product comprises a simulator of the RISC AR5 designed by Nayda G. Santiago. The simulator performs operations that are found in processors widely available at the market. Some of the core functionalities are: fetching and execution of instructions, arithmetical and logical operations, simulation of I/O ports and step by step execution while displaying the contents of all the registers and memory.

The flexibility provided by a high level language separates us from appreciating the behavior of components in a realistic environment. Even though overall knowledge of how a processor works was achieved, the abstraction layer is still very far. The design of such a system at gate level will surely engage us in further thinking about the underlying issues and structure involved with realistic environments, allowing us to fully grasp the necessary concepts required in Computer Engineering.

References

Allen, Ian D. "Teaching Ian! D. Allen." *Teaching*. N.p., n.d. Web. <http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt>.

Heuring, Vincent P., Harry F. Jordan, and Miles Murdocca. *Computer Systems Design and Architecture*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2004. Print.