A Mysterious-Simple-Computer Regi Suggested  
(AMRS)

BY:

Joseph Gabriel Gonzales

Justine Paul Guiao

Jovielene Lanorio

Bernieson Sahagun

CMSC 132 ST-1L

COMPUTER ARCHITECTURE

1. **Introduction and Statement of the Problem**
   1. **SYNTAX**

* ADD *param1* *param2*

Adds the two parameters *(param1* and *param2)* and saves the value in *param1.* If the value of loaded in *param1* after the computation is greater than 2 digits, overflow flag is set to 1. If not, overflow flag is set to 0.

* LOAD *param1* *param2*

Loads and sets a value for *param1* using *param2*. If the value loaded is greater than two digits, overflow flag is set to 1. If not, overflow flag is set to 0. Floating point values are not allowed.

* SUB *param1 param2*

Subtracts the two parameters *(param1* and *param2)* and saves the value in *param1.* If the value of loaded in *param1* after the computation is greater than 2 digits, overflow flag is set to 1. If not, overflow flag is set to 0.

* CMP *param1 param2*

Compares *param1* and *param2* by subracting *param2* from *param1*. Zero flag and negative flag are set to 0 by default. If the result is 0, zero flag is set to 1. If the result is less than 0, the negative flag is set to 1.

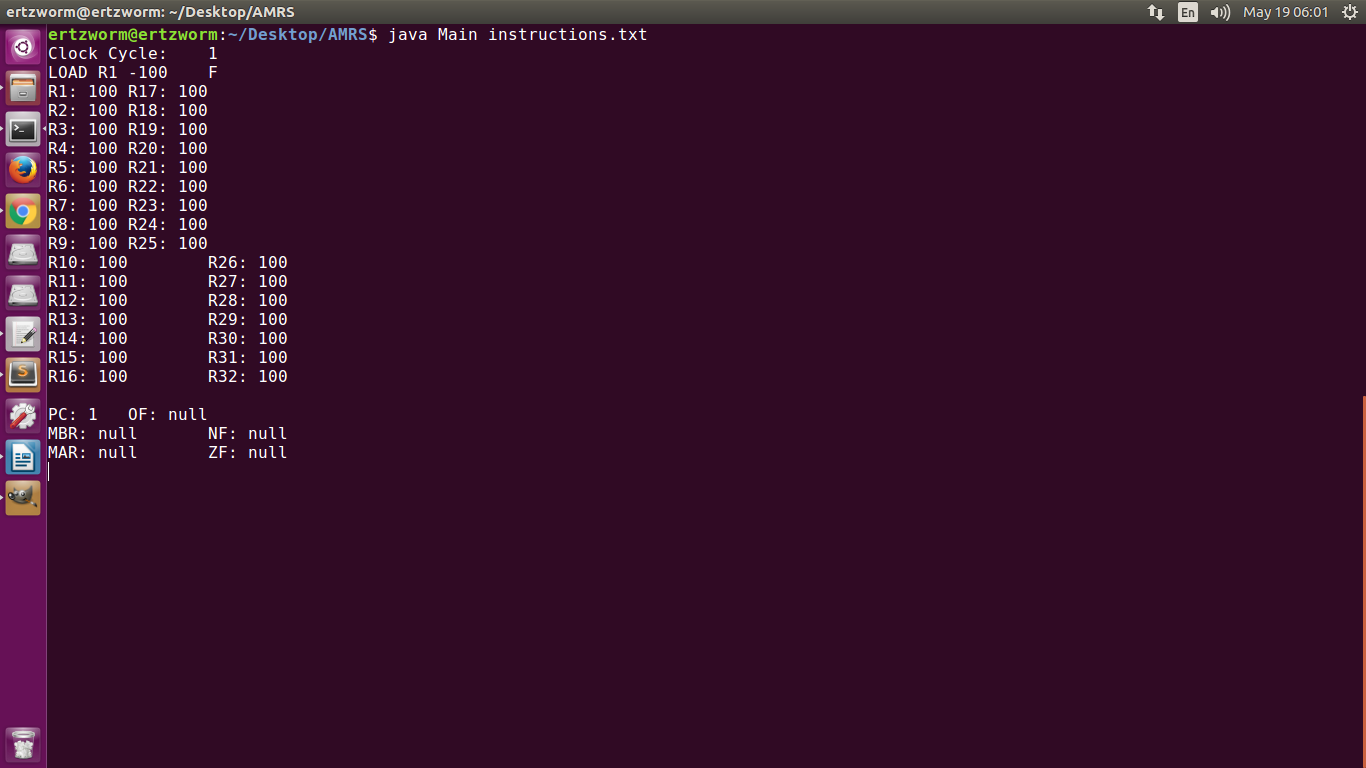
* 1. **ADDITIONAL SYNTAX ASSUMPTIONS**
* Instruction parameters inside the text file must be separated only by a space character. Other non-alphanumeric characters such as commas (,) are not allowed.
* Load instructions’ second parameter should always be an immediate value.
* Integer registers are initialized with the default value of 100.

1. **Pipelined Design**

* A 5-stage pipeline was implemented for this project. Each instruction is divided into Fetch, Decode, Execute, Memory Access, and Write Back.
* Stalls are to be called to avoid similar actions between instructions, or to partition instructions with dependencies.
  + i.e. Load *param1 param2* cannot be decoded in the same clock cycle with Load *param3* *param4*
* Fetch action can be called by the incoming instruction as the previous action uses Write Back.
* 32 integer registers ranging from R1 to R32 can be used for variable storage.
* Other registers such as Program Counter (PC), Memory Access Register (MAR), Memory Buffer Register (MBR), Overflow Flag (OF), Negative Flag (NF) and Zero Flag (ZF) can be used for incrementing instruction counter, storing value addresses, storing actual values, setting overflows, detecting negative values, and detecting zeroes respectively.

1. **Program Design**
   1. **Program Flow**

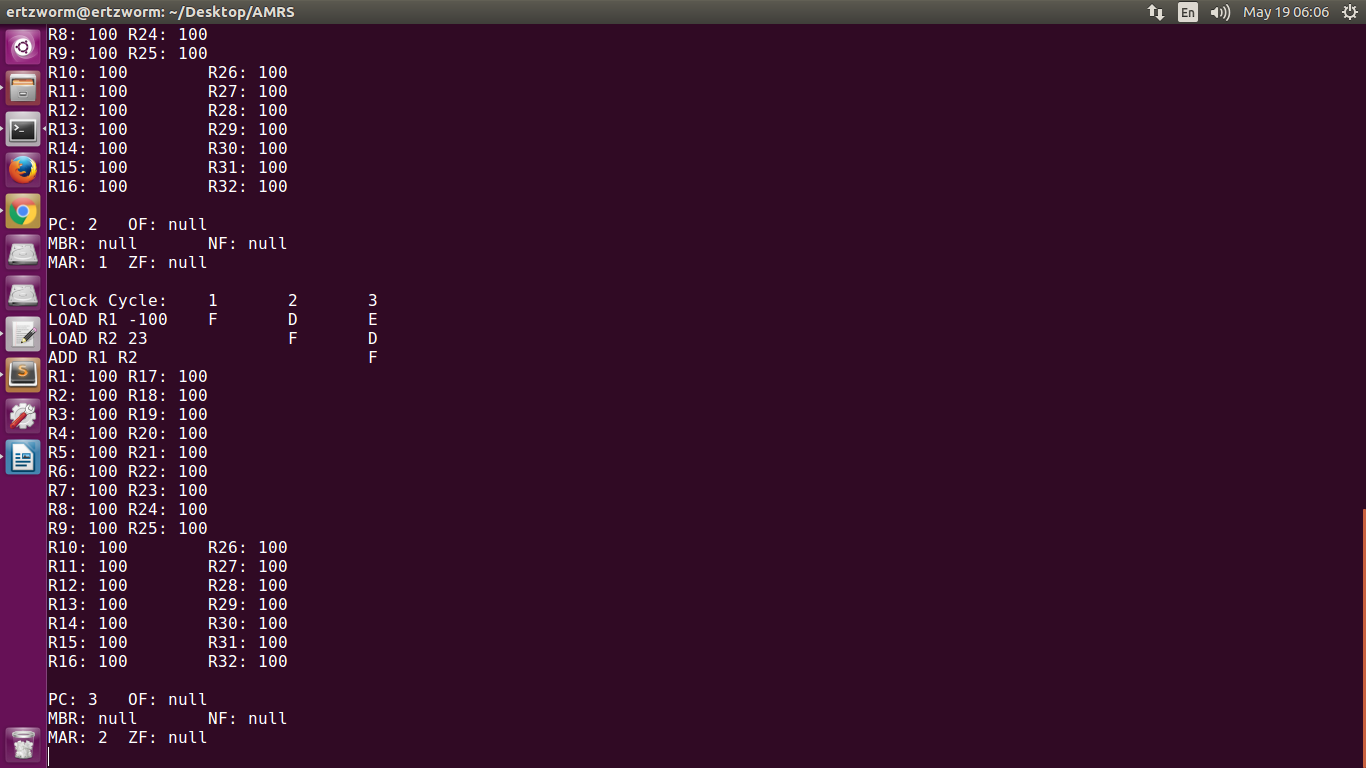
* The cycle of the program begins as follows:
  1. Instructions are parsed from a text file by the Parser class and are saved in a linked list. Parser class also detects the dependencies within the instructions.
  2. The total clock cycles for the entirety of the instructions computations is calculated preemptively.
  3. Special registers are instantiated with NULL values except for the Program Counter which is set with a “0”
  4. 32 integer registers are initialized with a temporary value of 100.
  5. An instance of the Scheduler class is initialized with parameters consisting of *instructions*, *dependencies*, *registers*, and *special* *registers*.
     1. Catches the parameters and sets them as local variables
     2. A counter for clock cycles is set as 0
     3. A queue for the actions (F, D, E, M, W) is initialized with null values
  6. Scheduler is started to begin execution of instructions
     1. Executes until it has no instructions left to perform
     2. Instructions are popped from a list and put in a queue for execution
        1. Dependencies are checked again if stalls are required. Current clock cycle is also incremented.
        2. Operations are performed.
        3. Registers are updated.
  7. **Program Output**



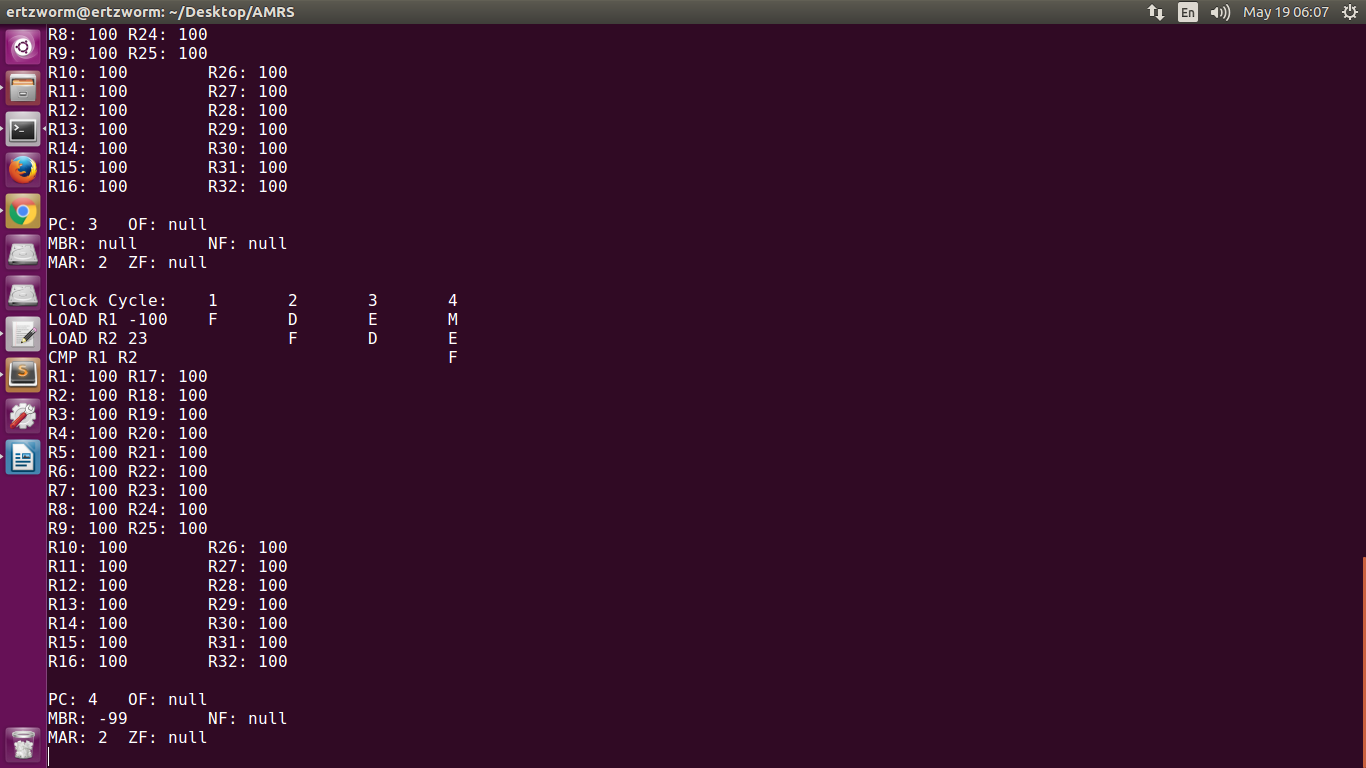
*Figure 1. First clock cycle*



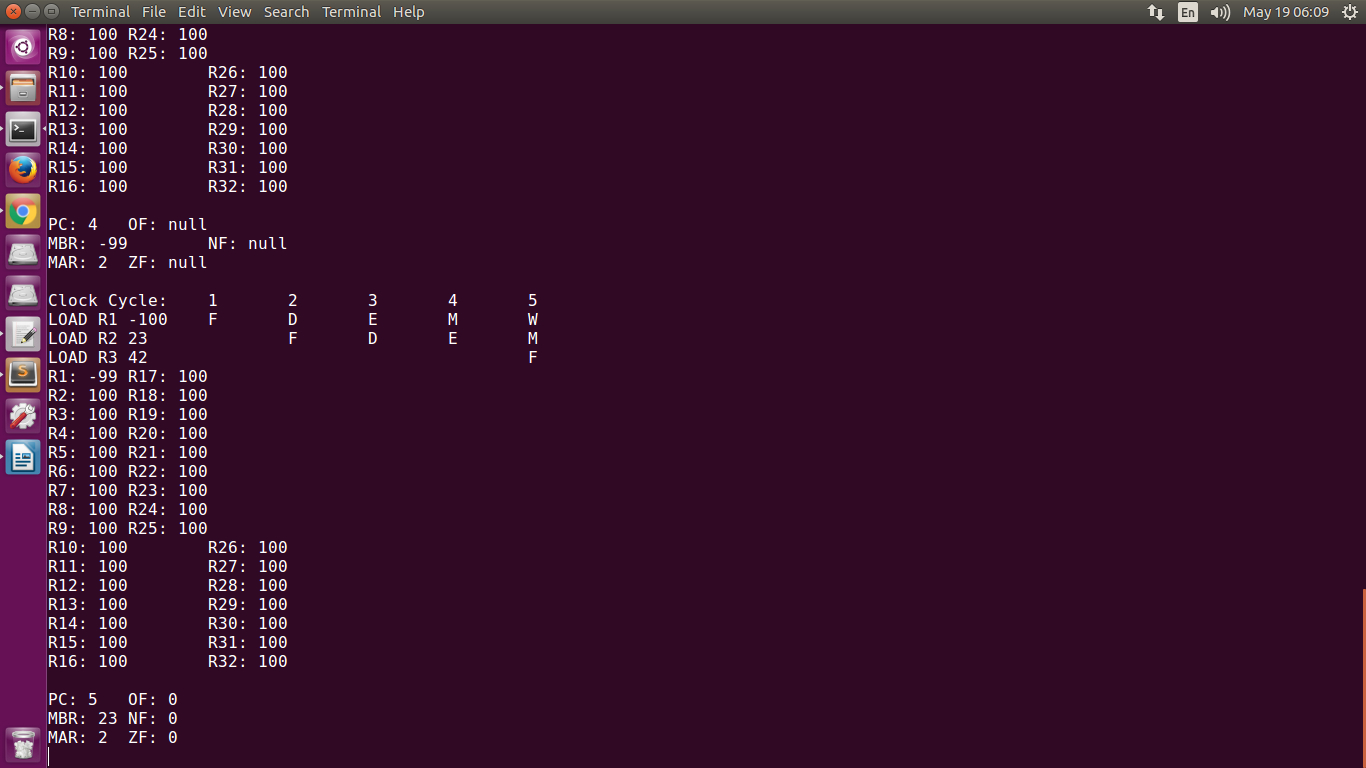
*Figure 1.1 As they ‘enter’ key is pressed, terminal shows the actions upon the next clock cycle*



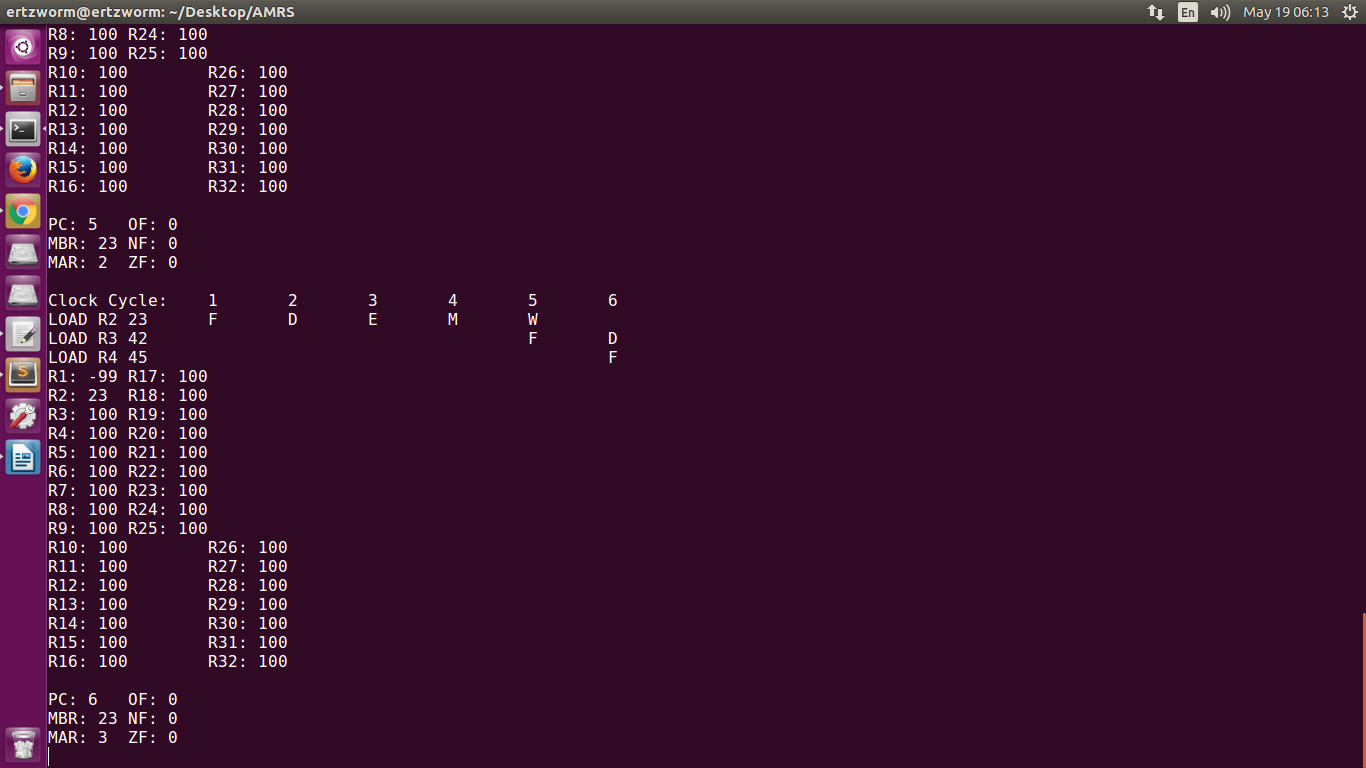
*Figure 1.2 Third clock cycle*

**

*Figure 1.3 Fourth clock cycle. Notice that MBR is already updated.*



*Figure 1.4 Fifth clock cycle*.



*Figure 1.5 Sixth Clock Cycle. Notice how R1 is updated to -99 as soon as the first clock cycle finishes executing.*

1. **User Manual**

* User must download the whole file system using git clone https://github.com/TheGabCode/AMRS.git
* User can change the values and instructions in “instructions.txt”
* Open terminal, compile and execute by entering the following commands
  + javac \*.java
  + java Main <filename>

1. **Conclusion**

* In the experimentation of this project, it has been observed how pipelining works and how it affects the efficiency of instruction execution in terms of scheduling. This project has addressed to portray a step-by-step process of how registers interact with the values of the instructions.
* It can be said that implemening a pipeline utilizes a CPU’s capability to process instructions thus making executions more efficient and lowering runtimes. Observing a step-by-step process of implementation gives a better understanding of how a CPU interacts with the registers.

1. **References**

* Teach-Sim Educational Simulators
  + The CPU Simulator, OS Simulator, Compiler
  + <http://www.teach-sim.com/>
* HASE – Computer Architecture Simulation Environment
  + http://www.icsa.inf.ed.ac.uk/research/groups/hase/