**" بسم الله الرحمن الرحيم "**

****

Student’s names: Razan Abdelrahman. ID: 1200531.

Sherin Ibdah. ID: 1200373.

Amr Halahla. ID: 1200902.

Project name: Simple implementation of Load, Store, and Add instructions.

Instructor’s name: Ayman Hroub.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Theory:

Computer systems contain three main building blocks: the central processing unit (CPU), memory, and input/output devices (I/O). These three components are connected together using the system bus. The most prominent items within the CPU are the registers: they can be manipulated directly by a computer program.

Within the CPU there is a set of general registers, and the number of them unit may vary from one processor to another.

1. One of the CPU registers is called as an accumulator AC or 'A' register. It is the main operand register of the ALU.it is used to store the result generated by ALU.

2. The data register (MDR) acts as a buffer between the CPU and main memory. It is used as an input operand register with the accumulator.

3. The instruction register (IR) holds the opcode of the current instruction.

4. The address register (MAR) holds the address of the memory in which the operand resides.

5. The program counter (PC) holds the address of the next instruction to be fetched for execution.

Additional addressable registers can be provided for storing operands and address.

\* Generic CPU Instruction Cycle:

* Fetch instruction.
* Decode instruction.
* Fetch operands from memory if necessary.
* Execute.
* Store result in memory if necessary.

# Objective:

\*Design control sequence for a simple computer using Verilog HDL. The simple computer is a very small computer to give you practice in the ideas of designing a simple CPU with the Verilog HDL.

\*To be familiar with the instruction execution cycle: fetch the instruction from memory, instruction decode, operand fetch and execution.

# Procedure:

The simple computer has a two-byte addressable memory with size of 256 bytes (i.e., one memory cell = 16 bits). The memory is synchronous to the CPU, and the CPU can read or write one cell (2 Bytes) in a single clock cycle. The memory can only be accessed through the Memory Address Register (MAR) and the Memory Buffer Register (MBR).

So our simple computer has 128 memory cells (the width for it is 16bits).

In our module, the states (0, 1) show how to fetch the instruction (Load, Store, Add) from memory cells that the contents of PC have loaded to MAR and increment the value of PC, then to read from memory address MAR , store the value of MAR in MBR , finally the contents of MBR are loaded into the IR.

Then in state (2) we decode the instruction by move 8bits from IR to the MAR.

After that in state (3) fetch operand is done. Operands from memory need to be fetched but the operands from registers need not to be fetched.

Finally in state (4), the instruction which stored in the IR will be executed.

This CPU has 16 general-purpose registers. So we implement the register file as a two-dimensional array each entry has size of 16 bits.

The opcodes included in our program are:

* 0011 LOAD Ri, M: loads the contents of memory location M into register Ri, where Ri is the number of the register.
* 1011 STORE Ri, M: Stores the contents of Ri into memory location M.
* 0111 ADD Ri, M: Adds the contents of memory location M to the contents of Ri, and stores the result in Ri.

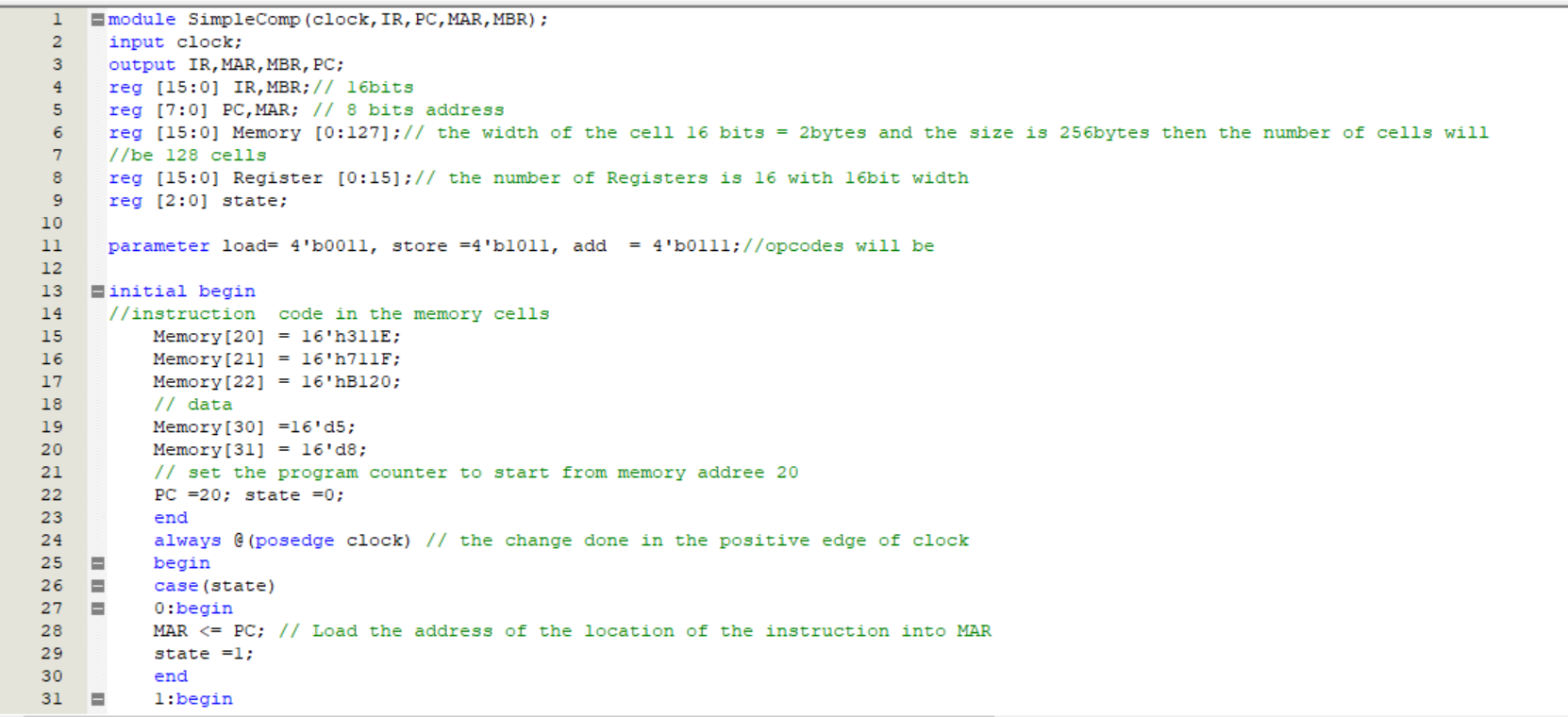
\*The instruction format:

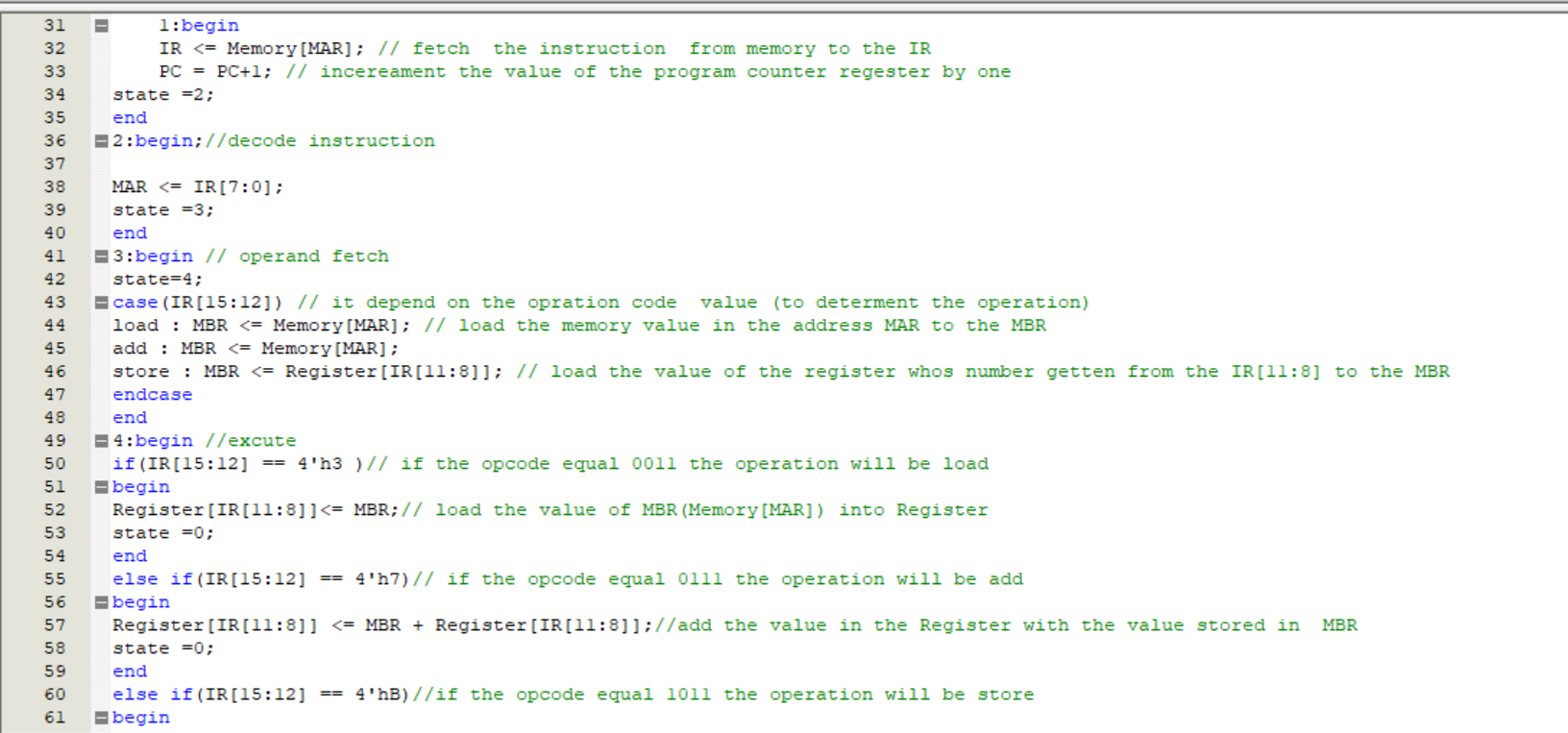
|  |  |  |
| --- | --- | --- |
| Opcode(4 bits) | Register (4 bits) | Memory Address (8 bits) |

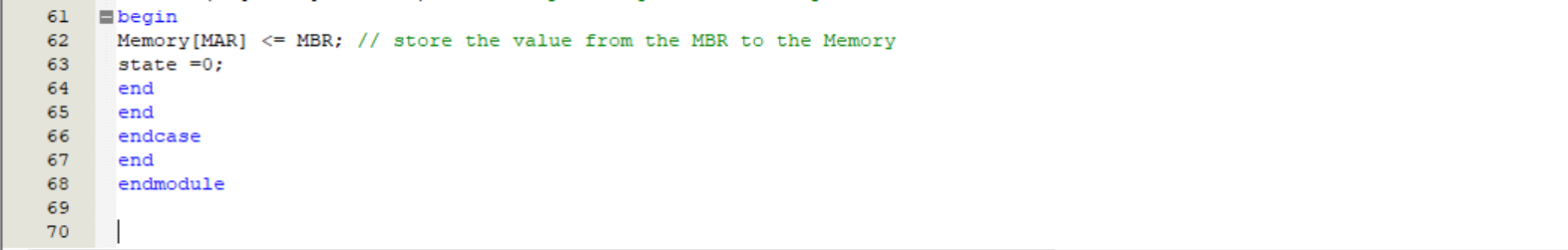
# Table of testing values:

|  |  |
| --- | --- |
| Memory address | Contents |
| 20 | **Load R1, [30] (instruction)** |
| 21 | **Add R1, [31] (instruction)** |
| 22 | **Store R1, [32] (instruction)** |
|  |  |
| 30 | **5 (data)** |
| 31 | **8 (data)** |
| 32 |  |

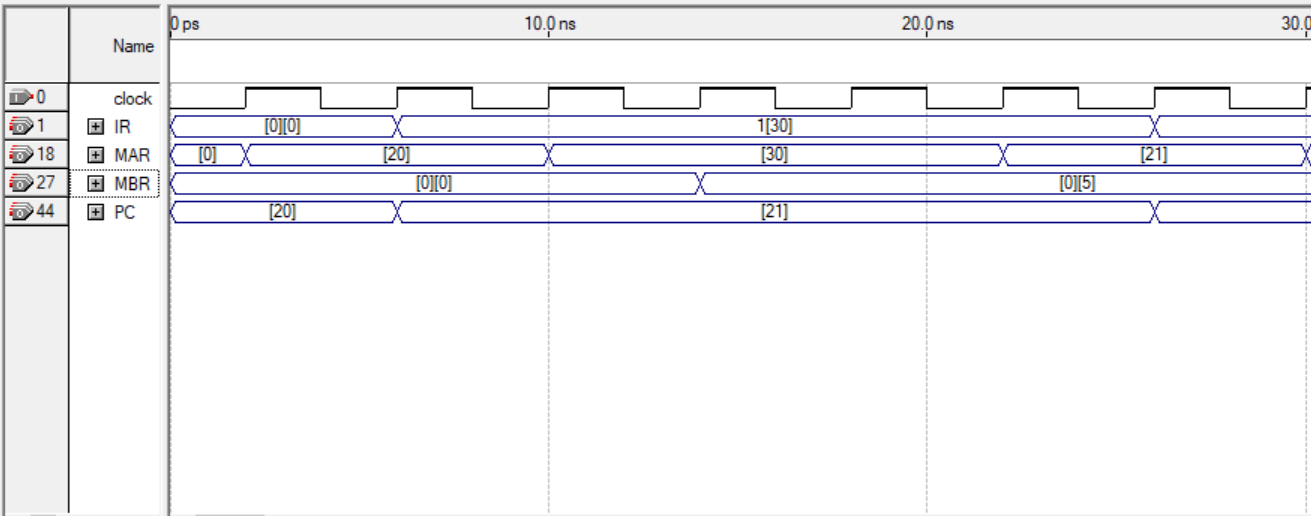
# The Verilog code for this simple computer:

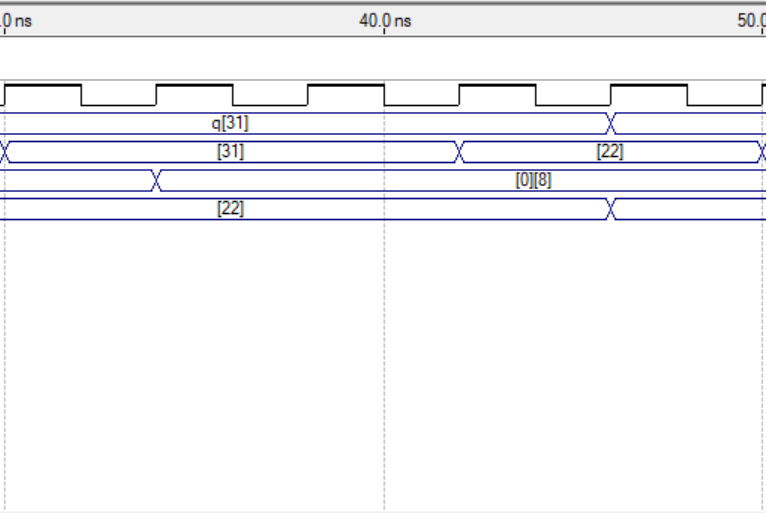






# The simulation result for this code:





DONE!