

Faculty of Engineering & Technology

Department of Electrical and Computer Engineering

**Digital Lab ENCS 211 EXP. No. 10**

**Simple Computer Simulation**

**Objective:**

In this experiment we are going to design the Verilog HDL control sequence for a simple computer (SIMCOMP). The SIMCOMP is a very small computer to give you practice in the ideas of designing a simple CPU with the Verilog HDL notation.

**Pre-Lab:**

1-Read the experiment

2-Do part one and part two of the procedure

**Background**

* **Basic Computer Model - Von Neumann Model**

Von Neumann 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, See figure one:

**Function of the Von Neumann Component:**

1. **Memory**: Storage of information (data/program)
2. **Processing Unit**: Computation/Processing of Information
3. **Input**: Means of getting information into the computer. e.g. keyboard, mouse
4. **Output**: Means of getting information out of the computer. e.g. printer, monitor
5. **Control Unit**: Makes sure that all the other parts perform their tasks correctly and at the correct time.



Figure 1: Von Neumann computer systems

* **General Registers:**

The number of registers in a processor unit may vary from one processor to another. Below are the general registers used by most processor:

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. This can be viewed as replacing the single accumulator by a set of registers. If the registers are used for many purpose, the resulting computer is said to have general register organization. In the case of processor registers, a registers is selected by the multiplexers that form the buses.

* **Communication Between Memory and Processing Unit**

Communication between memory and processing unit consists of two registers:

* Memory Address Register (MAR).
* Memory Data Register (MDR).
* To read,
	1. The address of the location is put in MAR.
	2. The memory is enabled for a read.
	3. The value is put in MDR by the memory.
* To write,
	1. The address of the location is put in MAR.
	2. The data is put in MDR.
	3. The **Write Enable** signal is asserted.
	4. The value in MDR is written to the location specified.
* **Generic CPU Instruction Cycle**

The generic instruction cycle for an unspecified CPU consists of the following stages:

1. **Fetch instruction:** Read instruction code from address in PC and place in IR. ( IR ← Memory[PC] )
2. **Decode instruction:** Hardware determines what the opcode/function is, and determines which registers or memory addresses contain the operands.
3. **Fetch operands from memory if necessary:** If any operands are memory addresses, initiate memory read cycles to read them into CPU registers. If an operand is in memory, not a register, then the memory address of the operand is known as the effective address, or EA for short. The fetching of an operand can therefore be denoted as Register ← Memory[EA]. On today's computers, CPUs are much faster than memory, so operand fetching usually takes multiple CPU clock cycles to complete.
4. **Execute:** Perform the function of the instruction. If arithmetic or logic instruction, utilize the ALU circuits to carry out the operation on data in registers. This is the only stage of the instruction cycle that is useful from the perspective of the end user. Everything else is overhead required to make the execute stage happen. One of the major goals of CPU design is to eliminate overhead, and spend a higher percentage of the time in the execute stage. Details on how this is achieved is a topic for a hardware-focused course in computer architecture.
5. **Store result in memory if necessary:** If destination is a memory address, initiate a memory write cycle to transfer the result from the CPU to memory. Depending on the situation, the CPU may or may not have to wait until this operation completes. If the next instruction does not need to access the memory chip where the result is stored, it can proceed with the next instruction while the memory unit is carrying out the write operation.

Below is an example of a full instruction cycle which uses memory addresses for all three operands.

  **mull x, y, product**

1. Fetch the instruction code from Memory[PC]
2. Decode the instruction. This reveals that it's a multiply instruction, and that the operands are memory locations x, y, and product.
3. Fetch x and y from memory.
4. Multiply x and y, storing the result in a CPU register.
5. Save the result from the CPU to memory location product.
* **Addressing Modes**

The term ***addressing modes*** refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand. Following are the main addressing modes that are used on various platforms and architectures.

**Our Simple Computer**

SIMCOMP has a two byte-addressable memory with size of 128byte. The memory is synchronous to the CPU, and the CPU can read or write a word in single clock period. The memory can only be accessed through the memory address register (**MAR**) and the memory buffer register (**MBR**). To read from memory, you use:

 **MBR <= Memory[MAR];**

And to write to memory, you use:

 **Memory[MA] <= MBR;**

The CPU has three registers -- an accumulator (**AC**), a program counter (**PC**) and an instruction register (**IR**). In addition, The SIMCOMP has only three instructions: **Load**, **Store**, and **Add**. The size of all instructions is 16 bits; all the instructions are single address instructions and access a word in memory.

Instruction Format

**The opcodes are:**

* **0011 LOAD M** // loads the contents of memory location **M** into the accumulator.
* **1011 STORE M** // stores the contents of the accumulator in memory location **M**.
* **0111 ADD M** // adds the contents of memory location **M** to the contents of the accumulator.

Procedure:

1- Study, write and simulate the SIMCOMP Verilog program (see the next page).

2- Add extra instruction (JUMP) to SIMCOMP

**JUMP M** jumps to location **M** in memory. Simulate the following program

Address Contents

|  |  |  |
| --- | --- | --- |
| 5 | Load | 9 |
| 6 | Add | 10 |
| 7 | Store | 11 |
| 8 | Jump | 6 |
| 9 | Data | 3 |
| 10 | Data | 2 |

.

**3-SIMCOMP2: Add register file**

Modify the instruction format so that SIMCOMP2 can handle four addressing modes and four registers. To this end, SIMCOMP is an **accumulator** machine which you can think of as a machine with one general-purpose register. Historically, many old computers were accumulator machines.

This new SIMCOMP2 has four 16-bit general purpose registers, R[0], R[1], R[2] and R[3] which replace the AC. In Verilog, you declare R as a bank of registers much like we do Memory:

**reg [15:0] R[0:3];**

And, since registers are usually on the CPU chip, we have no modeling limitations as we do with Memory - *with Memory we have to use the MAR and MBR registers to access MEM.* Therefore, in a load you could use R as follows:

**R[IR[9:8]] <= MBR;**

where the 2 bits in the IR specify which R register to set.

Modify the four instructions of the old SIMCOMP2 to the following new form:

**LOAD R[i],M** loads the contents of memory location **M** into R[i]. **STORE R[i],M** stores the contents of R[i] in memory location **M**.

**ADD R[i],R[j],R[k]** adds contents of R[j] and R[k] and places result in R[i].

**JUMP M** jumps to location **M** in memory.

To test your SIMCOMP2 design, perform the following program where PC starts at 10.

|  |  |  |
| --- | --- | --- |
| 3 | DATA | A |
| 4 | DATA | 6 |
| 10 | LOAD | R1,3 |
| 11 | LOAD | R2,4 |
| 12 | ADD | R1,R1,R2 |
| 13 | STORE | R1,5 |

**4-Add immediate addressing to the SIMCOMP2:**

If bit (IR[11])is a one in a Load , the last eight bits are not an address but an operand. The operand is in the range -128 to 127.

If immediate addressing is used in an LOAD, the operand is loaded into the register.

LoadI R1,8 R1 <- 8

Simulate the following test with hand written comments explaining what you are doing.

 **PC = 10**

**Memory [10] LoadI R1,3 // Load immediate**

**Memory [11] Store R1,4**

**Memory [12] LoadI R2,-4**

**Memory [13] Add R2,R2,R1**

**Memory [14] Store R2,5**

Note: your final machine should be able to correctly run the two "software" programs of all last two exercises. Be careful not to destroy the features of previous exercises. You should test this and include output in your hand in file to show that your **final** version of the SIMPCOMP2 works properly with the two programs.