BIRZEIT UNIVERSITY

Electrical and Computer Engineering

ENCS2380 – Computer Organization and Microprocessors- Fall 2022

Course Project (I)

Submission is only accepted on Moodle (itc.birzeit.edu)

Deadline: Thursday 19/01/2023 11:55PM.

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.

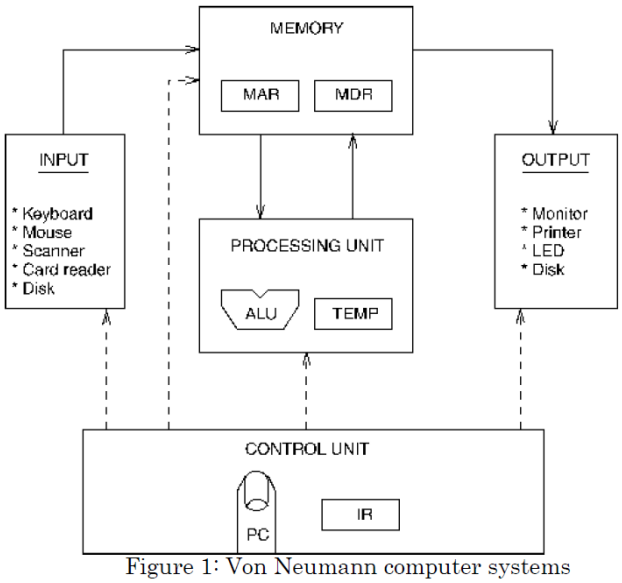


Figure 1: Von Neumann computer systems

- **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

---

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.

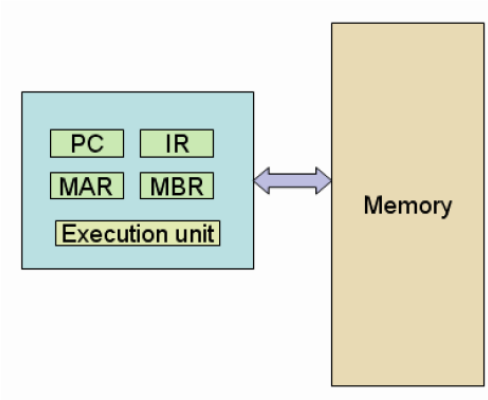
- Addressing Modes

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.

Example: Simple Accumulator computer implementation:

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

```
1  module SIMCOMP(clock, PC, IR, MBR, AC, MAR);
2   input clock;
3   output PC, IR, MBR, AC, MAR;
4   reg [15:0] IR, MBR, AC;
5   reg [11:0] PC,MAR;
6   reg [15:0] Memory [0:63];
7   reg [2:0] state;
8
9   parameter load = 4'b0011, store = 4'b1011, add=4'b0111;
10
11  initial begin
12     // program
13     Memory [10] = 16'h3020;
14     Memory [11] = 16'h7021;
15     Memory [12] = 16'hB014;
16
17     // data at byte addres
18     Memory [32] = 16'd7;
19     Memory [33] = 16'd5;
20
21     //set the program counter to the start of the program
22      PC = 10; state = 0;
23  end
24
25
26  always @(posedge clock) begin
27  case (state)
28  0: begin
29       MAR <= PC;
30       state=1;
31       end
32   1: begin // fetch the instruction from
33       IR <= Memory[MAR];
34       PC <= PC + 1;
35       state=2; //next state
36       end
37   2: begin //Instruction decode
38       MAR <= IR[11:0];
39       state= 3;
40       end
41   3: begin // Operand fetch
42       state =4;
43        case (IR[15:12])
44           load : MBR <= Memory[MAR];
45           add  : MBR <= Memory[MAR];
46           store: MBR<=AC;
47       endcase
48       end
49
50  4: begin //execute
51       if (IR[15:12]==4'h7) begin
52           AC<= AC+MBR;
53           state =0;
54       end
55       else if (IR[15:12] == 4'h3) begin
56           AC <= MBR;
57           state =0; // next state
58       end
59       else if (IR[15:12] == 4'hB) begin
60           Memory[MAR] <= MBR;
61           state = 0;
62       end
63      end
64  endcase
65  end
```

Project description:

In this project, we need to upgrade our accumulator computer described and implemented in the example above to a bit more complicated computer. The memory in this computer system is 16-bit cell memory with size of 128 cells (i.e. cell width = 16bits). The memory is synchronous to the CPU, and the CPU can read/write one cell in a single clock cycle. The memory can only be accessed through the memory address register (MAR) and the memory buffer register (MBR).

The CPU has a program counter (PC) register and an instruction register (IR). This CPU has eight 16-bit general-purpose registers R0-R7.

The 16-bit instruction format is as follow:

Opcode(3bits)	**Dst. Register (3bits)**	**mode (2bits)**	**Src. Register/ Memory Address (8bits)**

Addressing Modes bits : 00 => Direct memory address

01 => Register

10 => Register Indirect

11 => Constant

The opcodes are:

000 LOAD Ri, operand: loads register Ri with a memory cell of address operand/register content of address operand/ memory cell of address in register operand/or a constant integer operand.

001 STORE Ri, M: Stores the contents of **Ri** in memory location **M. (only mode=00 is supported with store instruction)**

010 ADD Ri, operand: Adds the contents of **Ri to the operand,** and store the result in **Ri.**

011 SUB Ri, M: subtract the contents operand from **Ri** and store the result in **Ri.**

0100 MUL Ri, M: Multiplies the contents of **Ri by the operand,** and store the result in **Ri.**

0101 DIV Ri, M: Divides the contents of operand by **Ri** and store the result in **Ri.**

Assume data is singed 16-bit integers in the sign-magnitude format as follow:

Sign (1bit)	Magnitude (15 bits)

Sign =0 => +ve numbers

Sign =1 => -ve numbers

Example:

To add memory cell of address 100 with memory cell pointed by register R1, and store the result in memory cell of address 102, we use the following program:

Assembly	Machine code
Load R0, [10]	000 000 00 00001010
Load R1, 11	000 001 11 00001011
Add R0, [R1]	010 000 10 00000001
Store R0, [12]	001 000 00 00001100

Requirements:

1) a- How many instructions this machine can support?

b- What is the range of unsigned and signed constant numbers this machine supports?

c- What is the length (in bits) of the following registers in this machine?

PC:

IR:

MAR:

MBR:

2) Simulate the following program by converting each instruction to corresponding machine code. Then store the machine code in memory starting from location 10:

Memory Address	**Contents**
10	Load R0, [20] (instruction)
11	Load R1,21
12	Add R0, [R1] (instruction)
13	load R1, [22] (instruction)
14	Sub R1, +8 (instruction)
15	Add R0,R1 (instruction)
16	Store R0, [23] (instruction)
20	6 (data)
21	4 (data)
22	13 (data)
23	0 (data)
24	0 (data)

Verify that it works correctly and the also verify that the result stored at address 24 is correct. Attach simulation waveform and the Verilog source file.

3) Assume A,B,C,D,E and Y are memory cells with addresses 30,31,32,33,34, and 35, respectively.

Given , $Y = \frac{A+B*C-1}{D-E}$,

a) Write assembly code for implementing the above arithmetic expression?

b) Convert the above assembly instructions into machine code and store them in the memory starting at address 10.

address	content
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	

c) Set PC=10 and simulate the above program. Verify that it works correctly and the result stored at memory variable Y is correct. Attach simulation waveform and the Verilog source file. **Assume A, B, C, D and E have the values -1, 3, 5, 8, and 4, respectively.**