BIRZEIT UNIVERSITY

Electrical and Computer Engineering

ENCS2380 – Computer organization and Microprocessors (Fall-2021)

Project (I)

Submissions are accepted only through Moodle (itc.birzeit.edu)

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.

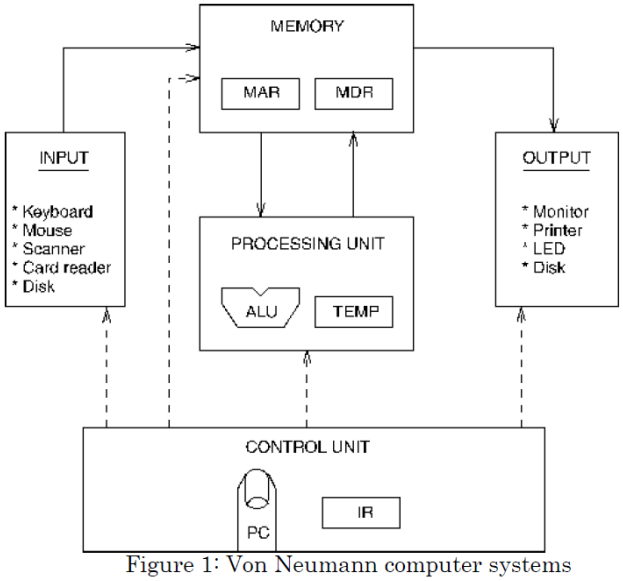


Figure 1: Von Neumann computer systems

- **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

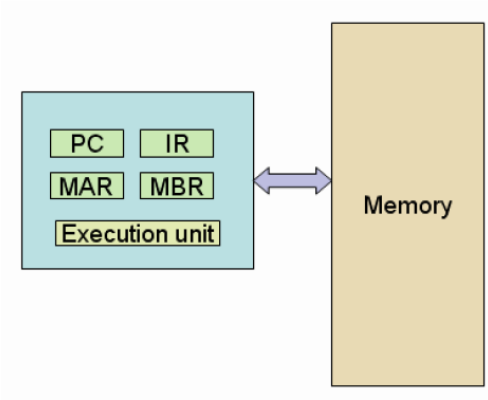
The term ***addressing modes*** refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand. Following are the main addressing modes that are used on various platforms and architectures.

Note: No submission for Part 0.

Part 0: Simple Accumulator Computer

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
```

(Part 1: Deadline: Monday 8/11/2021)

Part 1: Simple implementation of Load, Store, and Add instructions.

Objectives:

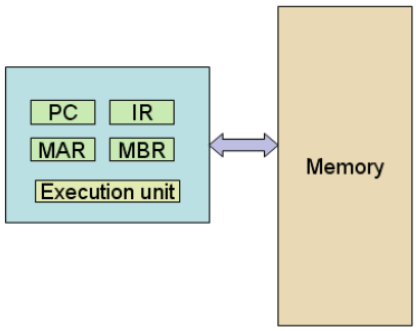
1- Design control sequence for a simple computer using Verilog HDL. The simple computer is a very small computer to give you practice in the ideas of designing a simple CPU with the Verilog HDL.
2- To be familiar with the instruction execution cycle: fetch the instruction from memory, instruction decode, operand fetch and execution.

As shown in the figure below, 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**).

To read from memory, you use: MBR ← Memory [MAR];

To write to memory, you use: Memory [MAR] ← [MBR];

The CPU has a program counter (PC) register and an instruction register (IR). This CPU has 16 general-purpose registers. You have to implement the register file as a two-dimensional array each entry has size of 16 bits. For example, to access register 0 you can use R[0] notation .You have to implement the flag register in one entry of this array.



This simple computer has only three instructions -- Load, Store, and Add. The size of all instructions is 16 bits with the following format:

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

The opcodes 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**.

Testing:

Simulate the following program by converting each instruction to the corresponding machine code. Then store the machine code in memory starting from location 20.

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	

(Part2: Deadline: Monday 22/11/2021)

Part 2: Add More Instructions and More Addressing Modes:

Objective: to be familiar with different addressing modes.

Update the instruction format by adding 3 bits to select the required addressing mode. The instruction format as following

Opcode (4)	Reg (4)	Operand (8)	Addressing mode (3)

Addressing mode field consist of 3 bits that determine the addressing mode of the operand field, as follow:
1-Direct Addressing (000): you have already implemented it in part 1.

2-Indirect Addressing (001): in this mode, memory cell pointed to by address field contains the address of (pointer to) the operand.

3-Immediate Addressing (010): in this mode, the operand is a part of the instruction format.

4-Register Addressing (011): in this mode, the operand is held in a register specified in the operand filed.

5-Stack Addressing (100): you should allocate part of the memory as a stack of size 20 bytes starting from location 0 (from entry 0 to entry 19) in the memory.

Task:

Modify the old simple computer to add the following new forms:

0011 LOAD Ri, M; loads the contents of memory location M into Ri, where Ri is the number of the register
0011 LOAD Ri, 8; set Ri to 8 (Immediate Addressing)
0011 LOAD Ri, [[M]]; use the contents of memory location M as a pointer to the operand then load it to Ri,

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.
0111 ADD Ri, Rj; Ri = Ri + Rj

1100 JUMP M; unconditional jump to location M in memory.
1101 CMP Ri, Rj; compare two registers and set zero flag if Ri = Rj
1110 SL Ri, C; applying logical shift left operation to Ri, such that, C is constant (Immediate Addressing)
1111 SR Ri, C; applying logical shift right operation to Ri,C is constant (Immediate Addressing)

0000 PUSH Ri; Add Ri to the top of the stack
0001 POP Ri; Ri = top of the stack then clear the top of the stack
When you convert each form to corresponding machine code, you can construct the instruction format as following:

1-Opcode (4): Opcode of each instruction, for example
LOAD Ri, 8 (0011)

2-Register and Operand: if the form has two operands, such as, LOAD Ri, M, use Register field for Ri and operand for M. For example,
Load R5, [3] → Register filed=0101 and operand =00011

If the form has one operand only, such as, JUMP 7

Operand=00111 and don’t care to Register filed.

3-Addressing mode (3): your job is finding the addressing mode for each form, for example
LOAD Ri, 8 (Immediate Addressing)

Testing:

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

Address	**Contents**
20	Load R1,[30]
21	Load R2,4
22	Add R1,R2
23	Store R1, [31]
30	5 (Data)
31	

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

Address	**Contents**
20	Load R1,3
21	Add R1,[31]
22	Load R2,6
23	CMP R1, R2
30	5 (Data)
31	

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

Address	**Contents**
20	Load R1,[[31]]
21	Shl R1,3
30	5 (data)
31	32(address)
32	3 (data)

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

Address	**Contents**
20	Load R1, [30]
	Push R1
21	Shr R1,1
22	Pull R1
23	Store R1, [33]
30	5 (data)
31	32 (address)
32	3 (data)
33	