**Summer Internship Project 1:**

**Adding CPU**

Name of the students with ID:

1. N190376 - L. Uma
2. N190753 - M. John David

**Introduction:**

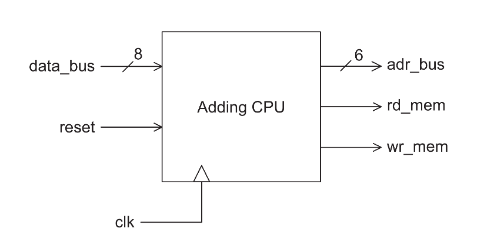
Adding CPU is simply a von Neumann model with memory accessing mechanism for instructions and data.

What is “von Neumann model”? The Von Neumann architecture consists of a single, shared memory for programs and data. The von Neumann computer model is based on a processor using instructions and data from a single memory.

**Differences between Von Neumann and Harvard Architecture :**

|  |  |
| --- | --- |
| **Von Neumann** | **Harvard** |
| * Uses a single memory for both instructions and data. * Single Bus is utilized for both instructions and data. * Simpler control unit design due to unified memory system * Generally easier to implement cost effective * Commonly used in general-purpose computers and systems where cost and simplicity are more critical. * Example: Most desktop and laptop computers. * Instructions and data cannot be fetched simultaneously due to the single bus. This is termed as Von Neumann bottleneck. | * Uses a separate memory for both instructions and data. * Uses separate buses for instructions and data, allowing simultaneous access to both instructions and data. * More Complex control unit design due to seperate memory systems * More expensive to implement due to separate buses and memory * Commonly used in specialized systems like microcontrollers, DSP’s where performance and efficiency are critical. * Example: Microcontrollers like PIC, AVR families. * Allows simultaneous fetching of instructions and data, which can result in faster execution and reduced bottleneck. |

The design of Adding CPU is at RT Level and has a separate Datapath and control unit.



**Details of processor functionality:**

Processor fetches instructions from its memory. Every Instruction has an opcode indicating the function it is supposed to perform. Such an operation may involve reading or writing data from memory or to the memory.

* As shown in above figure Adding CPU has a 6-bit address bus to address the memory for read and write operations.
* The 8-bit data\_bus of this CPU is used for data in and data out of the Processor.
* Control signals reset, rd\_mem and wr\_mem are used for resetting, memory read, and memory write operations.

**Opcodes & Instruction Format:**

Processor starts reading memory and fetches an 8-bit word from memory which consists of 2-bit opcode and 6-bit field. This 6-bit field can be either an immediate or a memory address.

The instruction Format and Opcodes of Adding CPU is as follows:

A screenshot of a computer

Description automatically generated

* **Load** – Processor Loads the Data from the Specified address of memory to the Accumulator Register (AC).
* **Store** – Stores the data from the AC to the specified addressed location in the memory.
* **Jump** – Loads the 6-bit address in to the Program Counter (PC) register.
* **Add** – Adds the immediate data to the contents of AC and stores the result back into the AC.

**Datapath:**

The main components of Datapath in adding CPU are Accumulator (AC), Program Counter (PC), Instruction Register (IR), and an ALU.

**PC** – Program Counter register keeps track of address from where the instruction is being fetched.

**AC -** AC register keeps data to operate on.

**IR** – Instruction register stores the most recent instruction that has been fetched.

**ALU** – Adder unit performs the addition operation and when it is store operation ALU just passes the data in AC in to dbus.

A diagram of a machine

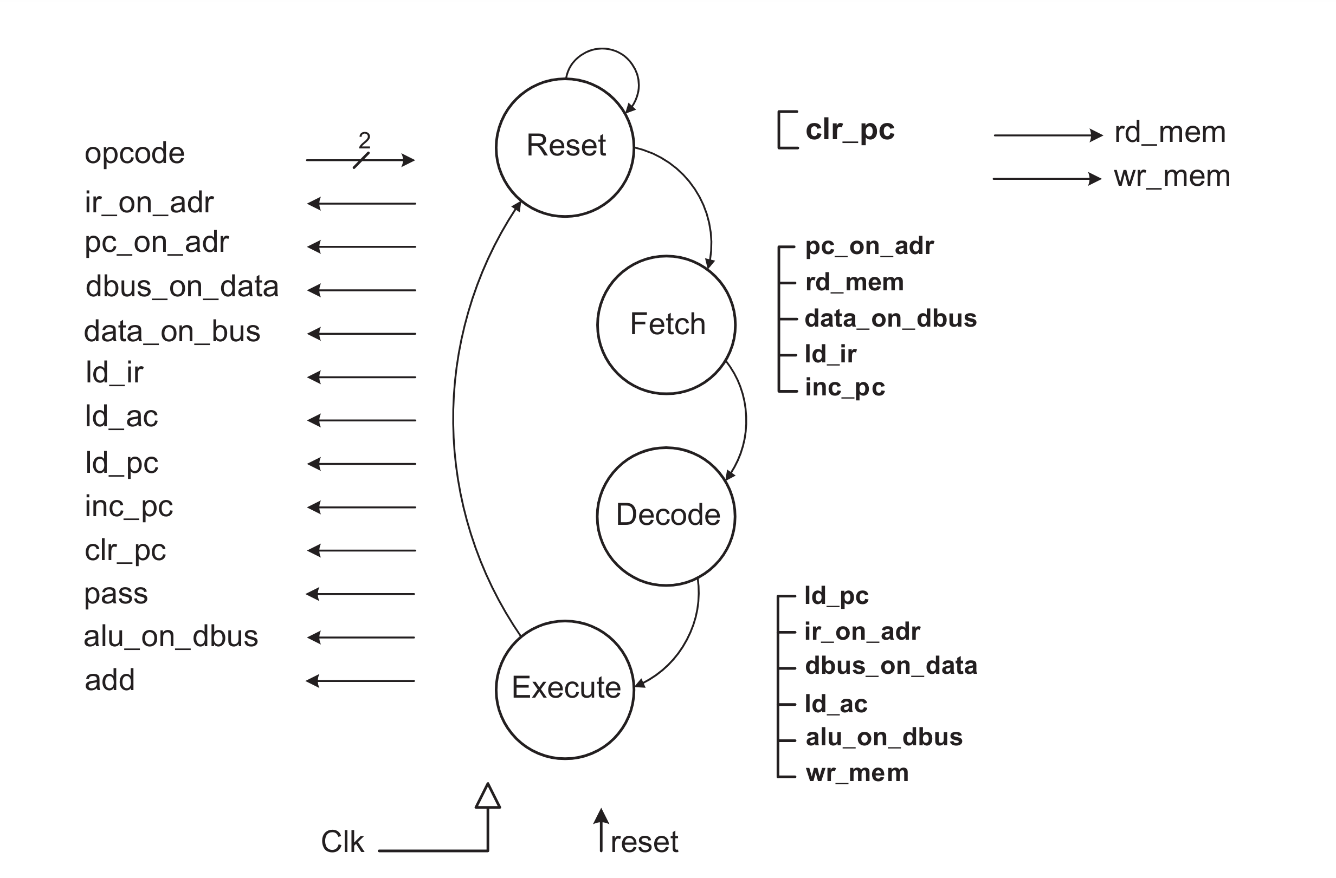
Description automatically generated

**Architectural Design of Adding CPU**

The above figure shows all the Datapath components and its required control signals.

**Control Unit:**

The controller part is a finite state machine (FSM) that looks at the opcode of the instruction in IR and decides on how data is to be routed and which signals has to give input for data path.



The Controller of Adding CPU simply has four states Reset, Fetch, Decode, and Execute. In Reset state which is executed initially clears the Program Counter (PC) register. In Fetch state instruction from memory is fetched to the Instruction register and increments PC to point to the next address. In Execute state depending up on the opcode control circuit generates the appropriate signals.

**Modelling the Adding CPU in Verilog HDL:**

**Datapath:**

//Instruction Register

module IR(data\_in,load,clk,data\_out);

input [7:0] data\_in;

input load, clk;

output reg [7:0]data\_out;

always@(posedge clk)

if (load)

data\_out <= data\_in;

endmodule

// Program Counter

module PC(data\_in,load,inc,clr,clk,data\_out);

input [5:0] data\_in;

input load, inc, clr,clk;

output reg [5:0]data\_out;

always@( posedge clk )

if( clr )

data\_out <= 6'b000000;

else if(load)

data\_out <= data\_in;

else if(inc)

data\_out <= data\_out + 1;

endmodule

// Accumulator Resisitor

module AC( data\_in,load,clk,data\_out);

input [7:0] data\_in;

input load, clk;

output reg [7:0] data\_out;

always @( posedge clk )

if( load )

data\_out <= data\_in;

endmodule

// ALU

module ALU(a,b,pass,add,alu\_out);

input [7:0] a, b;

input pass, add;

output reg [7:0]alu\_out;

always @(\*)

if (pass)

alu\_out = a;

else if (add)

alu\_out = a + b;

else

alu\_out = 0;

endmodule

// Complete DataPath....

module datapath(ir\_on\_adr, pc\_on\_adr, dbus\_on\_data, data\_on\_dbus, ld\_ir, ld\_ac, ld\_pc, inc\_pc, clr\_pc, pass, add, alu\_on\_dbus, clk,adr\_bus,op\_code,data\_bus);

// declaration input and output ports..

input ir\_on\_adr, pc\_on\_adr, dbus\_on\_data,

data\_on\_dbus, ld\_ir, ld\_ac, ld\_pc,

inc\_pc, clr\_pc, pass, add, alu\_on\_dbus,clk;

output [5:0] adr\_bus;

output [1:0] op\_code;

inout [7:0] data\_bus;

wire [7:0] dbus, ir\_out, a\_side, alu\_out;

wire [5:0] pc\_out;

IR ir( dbus, ld\_ir, clk, ir\_out );

PC pc( ir\_out[5:0], ld\_pc, inc\_pc, clr\_pc, clk, pc\_out );

AC ac( dbus, ld\_ac, clk, a\_side );

ALU alu( a\_side, {2'b00,ir\_out[5:0]}, pass, add, alu\_out );

assign adr\_bus = ir\_on\_adr ? ir\_out[5:0] : 6'bzzzzzz;

assign adr\_bus = pc\_on\_adr ? pc\_out : 6'bzzzzzz;

assign dbus = alu\_on\_dbus ? alu\_out : 8'bzzzzzzzz;

assign data\_bus = dbus\_on\_data ? dbus : 8'bzzzzzzzz;

assign dbus = data\_on\_dbus ? data\_bus : 8'bzzzzzzzz;

assign op\_code = ir\_out[7:6];

endmodule

**Control Unit Verilog Code:**

//Controller....

module Controlpath(input reset, clk, input [1:0] op\_code,

output reg rd\_mem, wr\_mem, ir\_on\_adr,pc\_on\_adr, dbus\_on\_data,data\_on\_dbus, ld\_ir, ld\_ac,ld\_pc, inc\_pc, clr\_pc,pass, add, alu\_on\_dbus);

parameter Reset=2'b00,Fetch=2'b01,Decode=2'b10,Execute=2'b11;

reg [1:0] present\_state, next\_state;

always @( posedge clk )

if( reset )

present\_state <= Reset;

else present\_state <= next\_state;

always@( present\_state or reset )

begin : Combination

rd\_mem=1'b0; wr\_mem=1'b0; ir\_on\_adr=1'b0; pc\_on\_adr=1'b0;

dbus\_on\_data=1'b0; data\_on\_dbus=1'b0; ld\_ir=1'b0;ld\_ac=1'b0; ld\_pc=1'b0; inc\_pc=1'b0;clr\_pc=1'b0; pass=0; add=0; alu\_on\_dbus=1'b0;

case ( present\_state )

Reset : begin next\_state = reset ? Reset : Fetch;

clr\_pc = 1;

end // End `Reset

Fetch : begin next\_state = Decode;

pc\_on\_adr = 1; rd\_mem = 1; data\_on\_dbus = 1;

ld\_ir = 1; inc\_pc = 1;

end // End `Fetch

Decode : next\_state = Execute; // End `Decode

Execute: begin next\_state = Fetch;

case( op\_code )

2'b00: begin

ir\_on\_adr = 1; rd\_mem = 1;

data\_on\_dbus = 1; ld\_ac = 1;

end

2'b01: begin

pass = 1;

ir\_on\_adr = 1; alu\_on\_dbus = 1;

dbus\_on\_data = 1; wr\_mem = 1;

end

2'b10: ld\_pc = 1;

2'b11: begin

add = 1; alu\_on\_dbus = 1; ld\_ac = 1;

end

endcase

end // End `Execute

default : next\_state = Reset;

endcase

end

endmodule

**The complete Adding CPU:**

In complete Adding CPU module both the Controlpath and datapath modules are instantiated.

Code:

//Complete Adding CPU Machine…

module addingcpu(reset,clk,adr\_bus,rd\_mem,wr\_mem,data\_bus);

input reset, clk;

output [5:0] adr\_bus;

output rd\_mem, wr\_mem;

inout [7:0] data\_bus;

wire ir\_on\_adr, pc\_on\_adr, dbus\_on\_data, data\_on\_dbus,ld\_ir,

ld\_ac, ld\_pc, inc\_pc, clr\_pc, pass, add, alu\_on\_dbus;

wire [1:0] op\_code;

Controlpath Cu( reset, clk, op\_code, rd\_mem, wr\_mem, ir\_on\_adr,

pc\_on\_adr, dbus\_on\_data, data\_on\_dbus, ld\_ir,

ld\_ac, ld\_pc, inc\_pc, clr\_pc, pass,

add, alu\_on\_dbus );

datapath dp( ir\_on\_adr, pc\_on\_adr, dbus\_on\_data, data\_on\_dbus,

ld\_ir, ld\_ac, ld\_pc, inc\_pc, clr\_pc, pass, add,

alu\_on\_dbus, clk, adr\_bus, op\_code, data\_bus );

endmodule

**Schematic of Adding CPU:**

**A screenshot of a computer

Description automatically generated**

**Testing the Adding CPU:**

For testing the Adding CPU we need a memory file to store instructions. So we created memory file mamed as “ instructions.mem ”. Each line in a **.mem** file represents a memory address and its corresponding data. The data can be in any format like binary, hexadecimal or decimal.

For testing the adding cpu we created the following instructions and stored them in hexadecimal format in instructions.mem file.

@0 20 🡺 Load 32 (i.e, load the data at 32nd location of mem in to AC)

@1 C4 🡺 Add 4 (i.e, add the immediate data with AC and store into AC)

@2 61 🡺Store 33 (i.e, Stores the AC data into 33rd location of mem)

@20 03 🡺 At the 32nd location of memory the data is 3.

**Test bench Code:**

//Test bench

module testbench;

reg reset=1, clk=0;

wire [5:0] adr\_bus;

wire rd\_mem, wr\_mem;

wire [7:0] data\_bus;

reg [7:0] memory[0:63]; // Declare memory array

initial begin

// Read the memory file into the array

$readmemh("instructions.mem", memory);

#25 reset=1'b0;

end

addingcpu UUT(reset, clk, adr\_bus, rd\_mem, wr\_mem, data\_bus);

always #10 clk = ~clk;

reg [7:0] mem\_data=8'b0;

reg control=0;

always@(posedge clk) begin : Memory\_Read\_Write

control = 1'b0;

#1;

if(rd\_mem) begin

mem\_data=memory[adr\_bus];

control = 1'b1;

end

if(wr\_mem) begin

#3 memory[adr\_bus]=data\_bus;

end

end

// Assigning( Copying) the mem\_data to the data\_bus

assign data\_bus = (control) ? mem\_data: 8'hZZ;

endmodule

**$readmemh**("instructions.mem", memory); This system task reads the hexadecimal data from **instructions.mem** and initializes the **memory** with the values. To verify we will check the address location mentioned in the store operation, which stores the result of add operation. So, we will verify the woking of Adding CPU, by whether that location is get stored with the addition result or not.

**Examples:**

Lets save the instructions.mem file with 6 instructions as shown below:

@0 0a 🡺 **Load** 10 (i.e, load the data at 10th location of mem in to AC)

@1 c5 🡺 **Add** 5 (i.e, add the immediate data(5) with AC and store into AC)

@2 4b 🡺**Store** 11 (i.e, Stores the AC data into 11th location of memory)

@3 86 🡺 **Jump** 6 (i.e, Jumps to the 6th location and execute instruction over there)

@7 4c 🡺 **Store** 12 (i.e, Stores the AC data into 12th location of memory)

@a 1 🡺 at 10th location in memory the data is 1.

@6 c2 🡺 at 6th location of memory the instruction is **add** 2.

Now, Lets see how to analyse the instructions and verify the results.

After PC get the reset it is updated to 0. So, the AddingCPU starts executing instructions from the 0th location of memory.

@0 0a 🡺 (0000 1010)2  Opcode bits = 00 and Address = 001010 = (10)10

we defined the instruction - **Load** 10 in 0th location. So, in **Fetch** state this get fetched into the IR register and decodes the instruction and in **Execute** state it goes to the mentioned address in this case it is ‘a’ (10) and Loads value at that address into AC. The data we mentioned in 10th Location is 1. So, after the execution of 1st instruction the AC has the value ‘1’ in it.

@1 c5 🡺 (1100 0101)2  Opcode bits = 11 and Immediate value = 000101 = (5)10

So, the second instruction is **add** 5. Fectching is same for every instruction ( that is getting the instruction in to IR). After Fectching based up on the Opcode bits the Execute state works. In this add instruction, immediate data is added with data in AC and result get stored back in to AC. So, after the execution of of 2nd instruction the AC should have the value 6 (1+5). To check whether the value is 6 or not, we store the AC in to memory and verifies that the addition had done correctly. This is way we check whether the processor working correctly or not.

@2 4b 🡺 (0100 1011)2  Opcode bits = 01 and Address = 001011 = (11)10

So, the 3rd instruction is **store** 11. The contents of AC get stored in the 11th location of memory.

@3 86 🡺 (1000 0110)2  Opcode bits = 10 and Immediate value = 000110 = (6)10

So, the 4th instruction is **jump** 6. That is the PC is updated to 6, and now processor jumps to 6th location of memory and executes the instruction over there. The instruction we specified in the 6th location is **add** 2. Previously the AC is 6, now by adding 2 it get updated with value 8. To check whether it is done or not we store the value in to 12th location which is our next instruction.

@7 4c 🡺 (0100 1100)2  Opcode bits = 01 and Immediate value = 001100 = (12)10

The instruction is **Store** 12. Stores the contents of AC register(8) into the 12th location of memory.

How we verify the above instructions is by looking at the 11th and 12th locations of memory it those locations are 6 and 8 respectively then we can conclude that our Adding CPU is working properly.

**Results:**

We can observe in waveforms that every time in Fetch state Databus has changing value, why because IR gets instruction from memory location through the Databus only, and also whenever we write data in to the memory. We can observe PC is initially 0 and whenever PC increases or Jump instruction is executed the address bus changes its value. All these changes we observe in the below:

**A screenshot of a computer

Description automatically generated**