

# German University in Cairo

## CSEN601 CA

### Project Report

Team Name: HPCA

Team Members :

- Mariam Adel Enany 43-2045
- Hoda Amr Elhemaly 43-5612
- Daila Walid 43-3745
- Aya Abdallah 43-18250
- Mayar Moawad 43-6645

Submission Date: 16/5/2020

In this project, we implemented a new microcontroller architecture using Java as a language. We assembled our own project by selecting different features that could help with the chosen architecture.

First, for the micro-architecture, we have chosen the Von Neumann architecture which has the following properties :

- It has one memory for data and instructions
- A single set of address/data buses between CPU and memory
- It allows only ONE memory fetch at a time

Secondly, we chose the size of the instruction memory and data memory. For each of the two memories the following memory size was picked (1024 x 16-bit ). The total number of registers used were 16 Registers.

As for the Instruction Format, we used Instruction set 1, which contained the following Instructions :

a) Arithmetic Instructions:

1. Add. 2. Add immediate. 3. Sub. 4. Multiply.

b) Logical Instructions:

1. And 2. Or immediate 3. Shift left logical. 4. Shift right logical.

c) Data Transfer Instructions:

1. Load word. 2. Store word

d) Conditional Branch Instructions:

1. Branch on not equal. 2. Branch on greater than

e) Comparison Instructions:

1. Set on less than

f) Unconditional Jump Instructions:

1. Jump

The instruction format that we created is as shown in fig(1).

Basic instruction formats

R	opcode	rs	rt	rd	funct	
	0      3    4	6      7	9     10	12    13	15	
I	opcode	rs	rt		immediate	
	0      3    4	6      7	9     10			15
J	opcode			address		
	0      3    4					15

fig(1)

Each instruction uses 16 bits in its format.

In the R-TYPE, 4 bits were used for (opcode) , 3 bits for( rs) , 3 bits for (rt) , 3 bits for (rd) and 3 bits for (funct).

In the I-TYPE, 4 bits were used for (opcode), 3 bits for( rs), 3 bits for (rt), and 6 bits for (immediate).

In J-TYPE, 4 bits were used for (opcode) and 12 bits for (address).

We have designed a specific opcode for each type to differentiate them.

For I-TYPE, we have three instructions ( Load Word(lw) with opcode "1000", Store Word(sw) with opcode "1011" , Add Immediate(addi) with opcode "1001"). R-TYPE opcode is "0000". J-TYPE opcode is "0010". Branch(BEQ) opcode "1100".

Regarding the ALUControl, the ALUOp of lw and sw is “00”, ALUOp of BEQ is “01” and ALUOp of R-TYPE is “10”.

To further differentiate between R-TYPE instructions, we look for funct. So there are four distinct bits for each instruction. The ALUOperation for ADD is "0011", for SUB is "0110", for AND is “0000”, for OR is “0001”, for SLT is “0111”, for MULT is “0010”, for SRL for “1110” and for SLL is “0101”.

The type of cache used is direct mapping cache. In this implementation, we are not using a specific replacement policy. The replacement is done according to the index of the address if it is not the same as the index at the cash we replace the existing address with the address we are searching for.

## **Code and flow**

In our projects we have three packages first one for components (the components that the PC use to run a program), second contains all stages that the instruction goes through to be executed in the last package is the processor package which contains Processor class, it's like a simulation of the Processor using our architecture and I will describe each package briefly below:

### **Components Package:**

We have 7 classes that represent a component and for each component. We have a class for ALU, Cache, Memory, PC, Register, Pipeline Register (used for the pipeline stage), and a Register file class that holds the 16 bits registers with their data.

there are common methods in cash and memory like write to and read from methods, in ALU we handle all arithmetic operations, comparisons and logic gate operations too, in PC class we use increment and getPc methods and in class Register, we have methods getRegister, setRegister, getData and setData of any register

By Accessing the components and using its methods we can do the stages that each instruction is supposed to go through.

### **Stages package:**

We have 5 stages ( Fetch, Decode, Execute, MemoryAccess, WriteBack), Each stage of them represents a cycle. Using pipelining to manage these stages.

### **Processor package:**

we have package processor in which the processor class handle all these stages for every instruction in a way that no more than one instruction can use the same component at the same time

And we will explain brief info about each stage below:

## **First of all InstructionFetch stage:**

After the Instruction is loaded from memory to cache, the instruction is fetched from the cache using method `readCashe` in class Cache and after that, the pc is incremented to point to the next instruction, And then setting the flag `fetch` that is used in the pipelining to True, as an indicator that the instruction finished the fetching process.

## **Second of all Decode:**

First, we set the flag `decode` to true to be as an indicator in the pipeline stage then we load the instruction to an array list named `a` to divide it part by part and take from it the opcode, rs, rt, funct, and the rest of the instruction.

Then according to the opcode, we determine it's type whether it's R-type, I-Type, or J-type.

For example, if it's R-type first we get the type of the registers used in the instruction by passing the registers to a method called `getType` that returns the type of the register used. After that, we need to make sure that the `Rd` register is not Zero Register as we can't modify it or write into it. if it's not then we specify in the decode the instruction format of each of the opcode, the `rs`, the `rt`, and the `funct`.

After that, we call the method `ControlUnit` to generate the control signals according to the opcode of each instruction.

The control Signals we used are :

```
RegWrite = 0;  
RegDst = 0; //RegDst indicates whether it's rd or rt register  
MemToReg = 0;  
MemRead = 0;  
MemWrite = 0;  
Branch = 0;  
Jump = 0;  
ALUOp = " ",
```

They're one-bit signals except for the `ALUOp` which is 2 bits.

So according to the opcode of each instruction, we decide whether to set each of these instructions to 1 or 0. After that, we call the method called

ALUControl that decides the 4-bit operation code for each instruction according to the ALUop and funct.

### **Execute :**

In Execute stage we first check the type of the instruction whether it's R-type, J-type or I-type according to its ALUop signal and also the opcode for example if the instruction is of I-type format and the ALUop is "00" then we know that it could be lw/sw or addi so the opcode indicates the type of the instruction and when we know the type of instruction in case of I-type we save it to ArrayList in order to access it in the pipelining and send it to the next stage.

If ALUop is "01" then we know that the instruction is of type Branch so we check the branching condition according to the instruction we have and if it satisfies the branching condition then we compute the branch address.

If it doesn't meet the conditions then we increment pc to point to the next instruction.

If the instruction is of R-type then we compute the result using the ALU class in package components and save to ArrayList to pass it to the next pipeline stage with its signals and registers used.

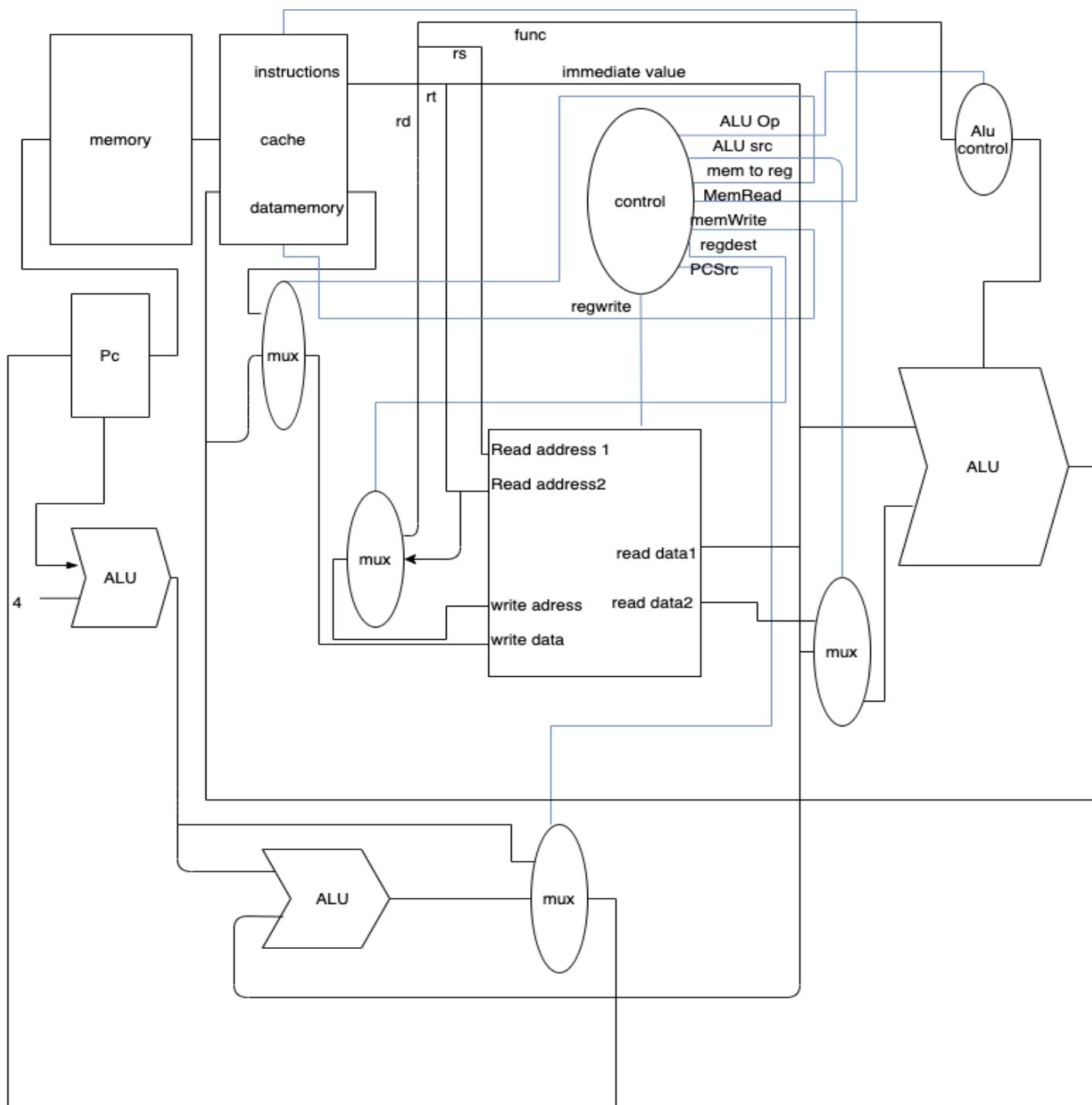
MemoryAccess: in memory access first we check the values of MemRead and MemWrite flags to know whether they're set on/off and according to the flags we decide whether to read or write.

In case of sw, we write into cache the data we've got it from the instruction. In the case of lw, we read the data from the cache and store it in ArrayList to pass it to the next pipeline stage.

WriteBack: in writeBack stage first we check RegWrite whether it's on or off and if it's on then we check first MemToReg flag whether it's on or off and if it's on then its lw and we get data from memory and write it to the register in the register file.

Else then if RegDst is on we write the data in the rd register in the register file.

## Data Path and all components Used:



This is our data path for the 16 bit register architecture , we have 6 components: memory , cache , ControlUnit , ALU ,Register File and PC(program counter)

First thing we upload all instructions of the program from the memory to the cache then we split every instruction to extract its OPcode , Registers , immediate value or address value if exists

Then we use our control unit to manage the different types of the instruction so first we have ALUOP which control the function the Alu would apply , it could be (add , sub , mult,...etc)

We have ALUsrc too which control the second input of the Alu that it could be a register value or some data we can extract from the cache

Then we have mem to register control which manages what should be written to the write address value in register file

Also We have MemRead and MemWrite which control the cash read and write methods

We have PCsrc control which manages us to go to the desired address in branch and jump instructions

Moreover we have RegDis which controls to which register the value should be saved

The last control we have is the write control which manages write operation to the RegisterFile

An ALU performs basic arithmetic and logic operations. Examples of arithmetic operations are addition, subtraction, multiplication, and division. Examples of logic operations are comparisons of values such as NOT, AND, and OR . Also we use ALU to increment the pc to go to the next instruction or go to the desired instruction if there is a branch or jump operations

The registerfile : is an array of registers the processor uses to swing data between memory and the functional units .

## Testing and Output:

First, the 16 bits instructions are loaded into memory and then loaded to Cache, and register file is also loaded when running the program.

```
instruction memory:
```

```
[0000001010011000, 0000100100100001, 0000100011101010, 0000101010110011, 0000110111111111, 0000001011011100, 0000010011100101, 0000001011011100, 0000
```

```
Register File before any changes:
```

```
[0: 0000000000000000 (DEC = 0, BIN = 0)][1: 0000000000000001 (DEC = 1, BIN = 1)][2: 0000000000000010 (DEC = 2, BIN = 10)][3: 0000000000000011 (DEC =
```

Second, we start with method fetch:

We fetch the instruction from the cache

```
fetching instruction: 1
miss
Instruction Fetched: 0000001010011000
Next PC: 0000000000000000
clock cycle: 1
```

Third, Decode stage:

```
***** decoding *****
.....
Type of Register rs: reserved for use by the assembler
Type of Register rt: Temporary registers
Type of Register rd: Temporary registers

opCode:0000|rs:001|rt:010|rd:011|funct:000

WB controls: MemToReg: 0 ,RegWrite: 1
MEM controls: MemRead: 0, MemWrite: 0, Branch: 0, Jump: 0
EX controls: RegDest: 1, ALUOp: 10, ALUSrc: 0
ALU OPERATION: 0011
***** finished decoding *****
```

Then fetching instruction 2:

```
.....
fetching instruction: 2
miss
Instruction Fetched: 0000100100100001
Next PC: 0000000000000001
clock cycle: 2
```

### Executing instruction 1:

```
executing instruction: 1
***** executing *****
.....
Operation Name: Add
ReadData1 in BIN: 0000000000000001 ,ReadData1 in DEC: 1
ReadData2 in BIN: 0000000000000010 ,ReadData2 in DEC: 2
ALUResult: 0000000000000011 in BIN, 3 in DEC
Z-Flag Value: 0
***** finished executing *****
```

### Decoding instruction 2:

```
Decoding instruction: 2
***** decoding *****
.....
Type of Register rs:
Type of Register rt:
Type of Register rd:

opCode:0000|rs:100|rt:100|rd:100|funct:001

WB controls: MemToReg: 0 ,RegWrite: 1
MEM controls: MemRead: 0, MemWrite: 0, Branch: 0, Jump: 0
EX controls: RegDest: 1, ALUOp: 10, ALUSrc: 0
ALU OPERATION: 0110
***** finished decoding *****
```

### Fetching instruction 3:

```
fetching instruction: 3
miss
Instruction Fetched: 0000100011101010
Next PC: 0000000000000010
clock cycle: 3
```

### Memory Access of instruction 1:

```
memory access stage of instruction: 1
***** memory access *****
.....
executing instruction: 2
***** executing *****
.....
Operation Name: SUB
ReadData1 in BIN: 000000000000100 ,ReadData1 in DEC: 4
ReadData2 in BIN: 000000000000100 ,ReadData2 in DEC: 4
ALUResult: 000000000000000 in BIN, 0 in DEC
Z-Flag Value: 1
***** finished executing *****
.....
```

### Decoding instruction 3:

```
Decoding instruction: 3
***** decoding *****
.....
Type of Register rs:
Type of Register rt: Temporary registers
Type of Register rd:

opCode:000|rs:100|rt:011|rd:101|funct:010

WB controls: MemToReg: 0 ,RegWrite: 1
MEM controls: MemRead: 0, MemWrite: 0, Branch: 0, Jump: 0
EX controls: RegDest: 1, ALUOp: 10, ALUSrc: 0
ALU OPERATION: 0000
***** finished decoding *****
.....
```

### Fetching instruction 4:

```
fetching instruction: 4
miss
Instruction Fetched: 0000101010110011
Next PC: 000000000000011
clock cycle: 4
.....
```

Write back of instruction 1:

```
write back stage of instruction: 1
***** Write Back *****

Destination Register: 3, Data to be Written: 3
Written Data in Register File: 3 in Decimal, 000000000000011 in Binary

***** finished Write Back *****
```

And so on for the rest of the instructions.

In the testing file, we've a total of 11 instructions testing different types of instructions operations we've and, the instructions are in a text file called "Program.txt".

The instruction format is in **Binary**.