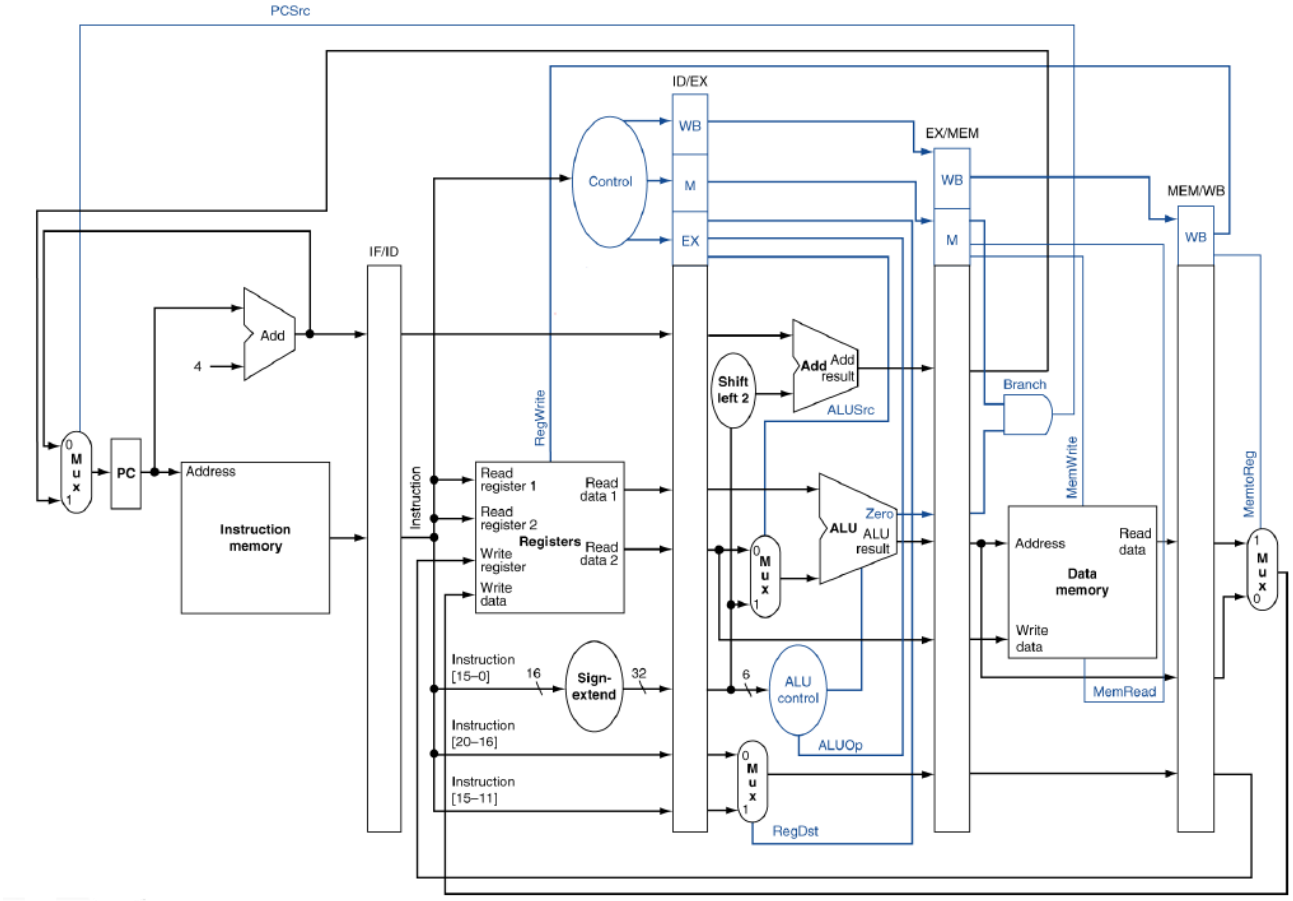
Computer Organization CO Lab 4

Architecture diagrams:



Hardware module analysis:

- Top module: Simple_Pipelined_CPU
- There are mainly 5 stage of pipelining:
  - **Instruction Fetch (IF)**
    - Program Counter: “count” for the next instruction that needs to do in the following clock cycle.
    - Adder: PC+4
    - Instruction memory: fetch the instructions from the text file according to the Program Counter.
  - **Instruction Decode (ID)**
    - Decoder: decode the opcode of the instruction and output to other modules for use.
    - Register File: performs the functions of getting the value from the register according to the instructions and storing the result of ALU into the register specified by the current instruction.
  - **Execution (EX)**

        - ALU Control: further decode the function code of the instruction and output to ALU for use.
        - Adder: add the value of Program Counter for the next instruction with the address stated by instruction[15:0], when is the instruction specified by branch operation.
        - Sign Extend: extend the 16-bit address stored in instruction into 32-bit.
        - multiplexor: for selecting the value of a register or the value extending the last 15 address of instruction according to the ALUSrc
        - ALU: perform some operations such as *and*, *or*, *addition*, *subtraction*, *set on less than*, *nor* and ***multiplication(simplified version)***
    - **Data Access (MEM)**
        - Data Memory: Read or Write the data from memory according to MemRead/ MemWrite.
    - **Write Back (WB)**
        - multiplexor: for selecting the input from Data Memory or the result of ALU according to MemtoReg (**simple modification: select the value from Data Memory when MemtoReg is 1, otherwise select the result of ALU**).
- **For each pipelined stage(exclude WB stage), there is a pipeline register storing the output including the control signals that need to pass to the next stage.**
    - **IF/ID:**
        - **output: PC+4, instruction**
    - **ID/EX:**
        - **output: PC+4, data of first and second register, result from Sign Extend, instruction[20-16] and instruction[15-11]**
        - **control: ALUOp, ALUSrc, RegDst, Branch, MemRead, MemWrite, RegWrite and MemtoReg**
    - **EX/MEM**
        - **output: The next relative instruction address need to be pass to the IF stage, Zero, the result of ALU, data of the second register, the address of the register to be written**
        - **control: Branch, MemRead, MemWrite, RegWrite, and MemtoReg**
    - **MEM/WB**
        - **output: the result of ALU, the output of the data memory, the address of the register to be written**
        - **control: RegWrite and MemtoReg**

Problems You Met and Solutions

1. Don't know how to store all the required, separated data into the only pipeline register for each stage.

   Solution: by textbook pg.347 stated that

   "*The register must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide,… the other three pipeline registers contain 128, 97, and 64 bits, respectively.* *(not include the data controls in this section)* "

   This description giving me the idea to store the values using the concatenation operator "{}".

```
Pipe_Reg #(.size(64)) IF_ID(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({IF_pc_add4,IF_instr}),
    .data_o({ID_pc_add4,ID_instr})
);
```

```
Pipe_Reg #(.size(128+10+10)) ID_EX(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({ID_pc_add4,
             ID_RSdata,
             ID_RTdata,
             ID_extend_out,
             ID_instr2,
             ID_instr3,
             ID_ALU_op_EX,
             ID_ALUSrc_EX,
             ID_RegDst_EX,
             ID_Branch_MEM,
             ID_MemRead_MEM,
             ID_MemWrite_MEM,
             ID_RegWrite_WB,
             ID_MemtoReg_WB
             }),
    .data_o({EX_pc_add4,
             EX_RSdata,
             EX_RTdata,
             EX_extend_out,
             EX_instr2,
             EX_instr3,
             EX_ALU_op,
             EX_ALUSrc,
             EX_RegDst,
             EX_Branch_MEM,
             EX_MemRead_MEM,
             EX_MemWrite_MEM,
             EX_RegWrite_WB,
             EX_MemtoReg_WB
             })
);
```

```
Pipe_Reg #(.size(97+5+5)) EX_MEM(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({EX_pc_branch,
             EX_zero,
             EX_ALU_result,
             EX_RTdata,
             EX_RDaddr,
             EX_Branch_MEM,
             EX_MemRead_MEM,
             EX_MemWrite_MEM,
             EX_RegWrite_WB,
             EX_MemtoReg_WB
            }),
    .data_o({MEM_pc_branch,
             MEM_zero,
             MEM_ALU_result,
             MEM_RTdata,
             MEM_RDaddr,
             MEM_Branch,
             MEM_MemRead,
             MEM_MemWrite,
             MEM_RegWrite_WB,
             MEM_MemtoReg_WB
             })
);
```

```
Pipe_Reg #(.size(64+5+2)) MEM_WB(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .data_i({MEM_ALU_result,
             MEM_dm_out,
             MEM_RDaddr,
             MEM_RegWrite_WB,
             MEM_MemtoReg_WB
            }),
    .data_o({WB_ALU_result,
             WB_dm_out,
             WB_RDaddr,
             WB_RegWrite,
             WB_MemtoReg
            })
);
```

Result:

test data 1:

```
Register=============================================

r0=     0, r1=     3, r2=     4, r3=     1, r4=     6, r5=     2, r6=     7, r7=     1
r8=     1, r9=     0, r10=    3, r11=    0, r12=    0, r13=    0, r14=    0, r15=    0
r16=    0, r17=    0, r18=    0, r19=    0, r20=    0, r21=    0, r22=    0, r23=    0
r24=    0, r25=    0, r26=    0, r27=    0, r28=    0, r29=    0, r30=    0, r31=    0

Memory==============================================

m0=     0, m1=     3, m2=     0, m3=     0, m4=     0, m5=     0, m6=     0, m7=     0
m8=     0, m9=     0, m10=    0, m11=    0, m12=    0, m13=    0, m14=    0, m15=    0
r16=    0, m17=    0, m18=    0, m19=    0, m20=    0, m21=    0, m22=    0, m23=    0
m24=    0, m25=    0, m26=    0, m27=    0, m28=    0, m29=    0, m30=    0, m31=    0
```

test data 2:

modify the instruction as below:

```
addi $1,$0,16
addi $3,$0,8
nop
addi $2,$1,4
sw $1,4($0)
lw $4,4($0)
nop
nop
sub $5,$4,$3
add $6,$3,$1
addi $7,$1,10
addi $9,$0,100
nop
and $8,$7,$3
```

and get the result:

```
Register=============================================

r0=     0, r1=     16, r2=    20, r3=    8, r4=     16, r5=    8, r6=     24, r7=    26
r8=     8, r9=     100, r10=  0, r11=    0, r12=    0, r13=    0, r14=    0, r15=    0
r16=    0, r17=    0, r18=    0, r19=    0, r20=    0, r21=    0, r22=    0, r23=    0
r24=    0, r25=    0, r26=    0, r27=    0, r28=    0, r29=    0, r30=    0, r31=    0

Memory==============================================

m0=     0, m1=     16, m2=    0, m3=     0, m4=     0, m5=     0, m6=     0, m7=     0
m8=     0, m9=     0, m10=    0, m11=    0, m12=    0, m13=    0, m14=    0, m15=    0
r16=    0, m17=    0, m18=    0, m19=    0, m20=    0, m21=    0, m22=    0, m23=    0
m24=    0, m25=    0, m26=    0, m27=    0, m28=    0, m29=    0, m30=    0, m31=    0
```

Summary:

This Lab implementing the Pipelined CPU that includes IF, ID, EX, MEM, and WB stage is relatively easy for me as it only implements the design according to the architecture in the textbook.