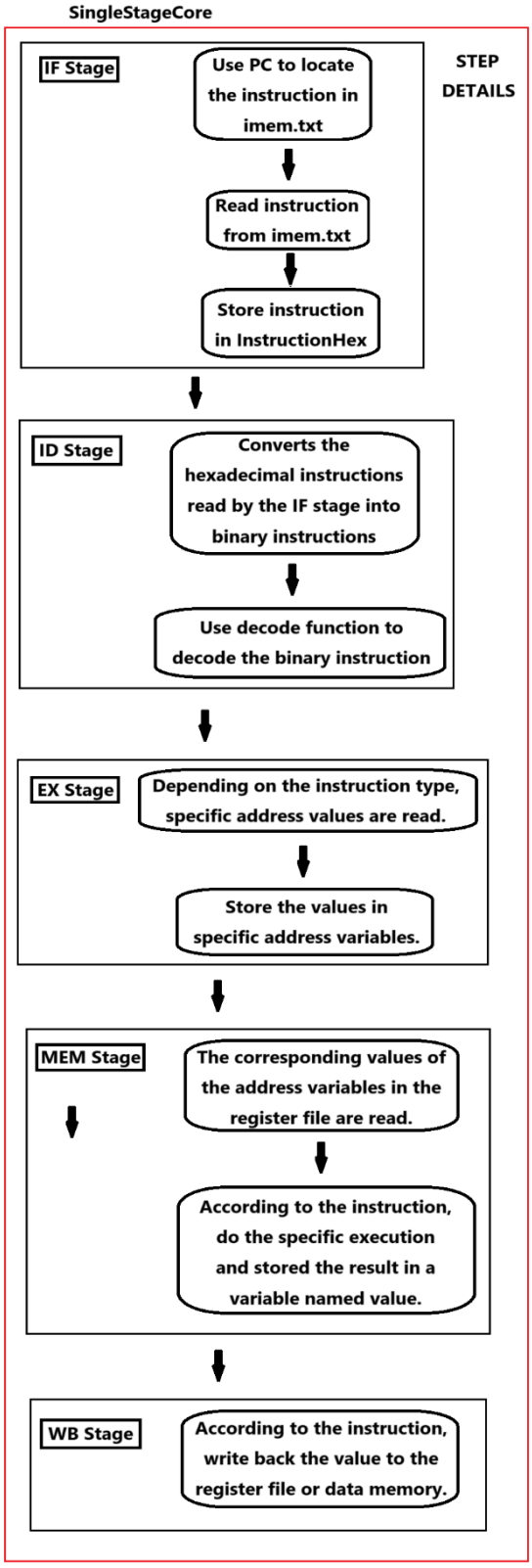
Phase 2 report

Task 1:

Single stage:

The schematic for a single stage processor is as following:



The detailed code in my implementation is as follows.

class InsMem: it is used to access instructions from imem.txt

```
class InsMem(object):  1 个用法
    def __init__(self, name, ioDir):
        self.id = name
        with open(ioDir + "\\imem.txt") as im:
            self.IMem = [data.replace( _old: "\n",  _new: "") for data in im.readlines()]

    def readInstr(self, ReadAddress):  2 用法
        #read instruction memory
        #return 32 bit hex val
        indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
        selected_strings = [self.IMem[i] for i in indices_to_select]
        for i in range(4):
            selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
        instruction = ''.join(selected_strings)
        return instruction

```

class DataMem: it is used to access and write back data memory.

```
class DataMem(object):  2 用法
    def __init__(self, name, ioDir):
        self.id = name
        self.ioDir = ioDir
        self.DMem = ["00000000"]*MemSize
        with open(ioDir + "\\dmem.txt") as dm:
            self.DMem = [data.replace( _old: "\n",  _new: "") for data in dm.readlines()]
        self.DMem.extend(["00000000"] * (MemSize - len(self.DMem)))

    def readDataMem(self, ReadAddress):  2 个用法 (2 个动态)
        #read data memory
        #return 32 bit hex val
        indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
        selected_strings = [self.DMem[i] for i in indices_to_select]
        #for i in range(4):
        #    selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
        DataMem = ''.join(selected_strings)
        is_negative = DataMem[0] == '1'
        if is_negative:
            inverted_binary = ''.join('1' if b == '0' else '0' for b in DataMem)  # 取反
            positive_value = int(inverted_binary, 2) + 1  # 加1，得到正数值
            DataMem = -positive_value  # 加符号
        else:
            DataMem = int(DataMem, 2)
        print("data in mem is "+str(DataMem))
        return DataMem

    def writeDataMem(self, Address, WriteData):  2 个用法 (2 个动态)
        # write data into byte addressable memory
        if WriteData < 0:
            binary_str = bin((1 << 32) + WriteData)[2:]  # 加 2^32 实现补码
        else:
            binary_str = bin(WriteData)[2:]
        binaryData = binary_str.zfill(32)
        binary_8bit_Data = [binaryData[i:i + 8] for i in range(0, 32, 8)]
        required_lines = Address + 4
        while len(self.DMem) < required_lines:
            self.DMem.append("\n")
        for i in range(4):
            self.DMem[Address + i] = binary_8bit_Data[i]

    def outputDataMem(self):  2 用法
        resPath = self.ioDir + "\\" + self.id + "_DMEMResult.txt"
        print(self.DMem)
        with open(resPath, "w") as rp:
            i = 0
            for data in self.DMem:
                if data != "\n" and i != len(self.DMem)-1:
                    rp.writelines(str(data)+"\n")
                else:
                    rp.writelines(str(data))
                i += 1

class RegisterFile(object):  1 个用法
```

class RegisterFile: it is used to read and modify the registers' values.

```python
class RegisterFile(object):  1 个用法
    def __init__(self, ioDir):
        self.outputFile = ioDir + "RFResult.txt"
        binary_str = format(0, '032b')
        self.Registers = [binary_str for i in range(32)]

    def readRF(self, Reg_addr):  21 用法
        DataReg = self.Registers[Reg_addr]

        is_negative = DataReg[0] == '1'
        if is_negative:
            inverted_binary = ''.join('1' if b == '0' else '0' for b in DataReg)  # 求反
            positive_value = int(inverted_binary, 2) + 1  # 加1，得到正数值
            DataReg = -positive_value  # 加符号
        else:
            DataReg = int(DataReg, 2)

        return DataReg

    def writeRF(self, Reg_addr, Wrt_reg_data):  4 用法
        if Wrt_reg_data < 0:
            binary_str = bin((1 << 32) + Wrt_reg_data)[2:]  # 加 2^32 实现补码
        else:
            binary_str = bin(Wrt_reg_data)[2:]
        binaryData = binary_str.zfill(32)
        self.Registers[Reg_addr] = binaryData

    def outputRF(self, cycle):  2 用法
        op = ["-"*70+"\n", "State of RF after executing cycle:" + str(cycle) + "\n"]
        op.extend([str(val)+"\n" for val in self.Registers])
        if(cycle == 0): perm = "w"
        else: perm = "a"
        with open(self.outputFile, perm) as file:
            file.writelines(op)
```

The decode function used in step is as follows: it is used to decode the binary instruction.

```python
    def decode(self, instruction):  1 个用法
        instructionInfo = {}
        opcode = instruction[-7:]
        op_R = ('0110011','0111011')
        op_I = ('0010011','0000011','0001111','0011011','1100111','1110011')
        op_S = '0100011'
        op_B = '1100011'
        op_U = '0110111'
        op_J = '1101111'
        if opcode in op_R:
            self.instructionInfo_R['func7'] = instruction[:7]
            self.instructionInfo_R['rs2'] = instruction[7:12]
            self.instructionInfo_R['rs1'] = instruction[12:17]
            self.instructionInfo_R['func3'] = instruction[17:20]
            self.instructionInfo_R['rd'] = instruction[20:25]
            self.instructionInfo_R['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_R
        elif opcode in op_I:
            self.instructionInfo_I['imm[11:0]'] = instruction[:12]
            self.instructionInfo_I['rs1'] = instruction[12:17]
            self.instructionInfo_I['func3'] = instruction[17:20]
            self.instructionInfo_I['rd'] = instruction[20:25]
            self.instructionInfo_I['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_I
        elif opcode == op_S:
            self.instructionInfo_S['imm[11:5]'] = instruction[:7]
            self.instructionInfo_S['rs2'] = instruction[7:12]
            self.instructionInfo_S['rs1'] = instruction[12:17]
            self.instructionInfo_S['func3'] = instruction[17:20]
            self.instructionInfo_S['imm[4:0]'] = instruction[20:25]
            self.instructionInfo_S['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_S
        elif opcode == op_B:
            self.instructionInfo_B['imm[12,10:5]'] = instruction[:7]
            self.instructionInfo_B['rs2'] = instruction[7:12]
            self.instructionInfo_B['rs1'] = instruction[12:17]
            self.instructionInfo_B['func3'] = instruction[17:20]
            self.instructionInfo_B['imm[4:1,11]'] = instruction[20:25]
            self.instructionInfo_B['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_B
        elif opcode == op_U:
            self.instructionInfo_U['imm[31:12]'] = instruction[:20]
            self.instructionInfo_U['rd'] = instruction[20:25]
            self.instructionInfo_U['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_U
        elif opcode == op_J:
            self.instructionInfo_J['imm[20,10:1,11,19:12]'] = instruction[:20]
            self.instructionInfo_J['rd'] = instruction[20:25]
            self.instructionInfo_J['opcode'] = instruction[-7:]
            instructionInfo = self.instructionInfo_J
        elif opcode == '1111111':
            self.state.IF["nop"] = True
        else:
            print(f"something wrong with the instruction decoding!!!")

        print("SS_OPCODE: " + opcode)
        return instructionInfo
```

The step is implemented as follows: it is the main working step during a cycle.

```
    def step(self):  1 个用法
        # Your implementation
        print("this is cycle: "+ str(self.cycle))
        rs1 = 0
        rs2 = 0
        rd = 0
        imm = 0
        value = 0
        address = 0
        op_R = ('0110011', '0111011')
        op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
        op_S = '0100011'
        op_B = '1100011'
        op_U = '0110111'
        op_J = '1101111'

        #IF stage
        if self.state.IF["nop"]:
            self.halted = True
            self.instructionCount += 1
        if self.state.IF["nop"] == False:
            InstructionHex = imem.readInstr(self.PC*4)
        #ID stage
        if self.state.IF["nop"] == False:
            InstructionBin = bin(int(InstructionHex, 16))[2:].zfill(32)
            InstructionInfo = self.decode(InstructionBin)
        #EX stage
        if self.state.IF["nop"] == False:
            if InstructionInfo['opcode'] in op_R:
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                rd = int(InstructionInfo['rd'], 2)

            elif InstructionInfo['opcode'] in op_I:
                rs1 = int(InstructionInfo['rs1'], 2)
                rd = int(InstructionInfo['rd'], 2)
                imm = int(InstructionInfo['imm[11:0]'], 2)

            elif InstructionInfo['opcode'] == op_S:
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                imm = int(InstructionInfo['imm[4:0]'], 2)
                address = rs1 + imm

            elif InstructionInfo['opcode'] == op_B:
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                imm_binary = (InstructionInfo['imm[12,10:5]'][0] + InstructionInfo['imm[4:1,11]'][4] +
                              InstructionInfo['imm[12,10:5]'][1:7] + InstructionInfo['imm[4:1,11]'][:4] + '0')
                imm = int(imm_binary, 2)
                # Sign extension for 13-bit immediate (if the 13th bit is 1, the value is negative)
                if imm_binary[0] == '1':  # Check if the highest bit (sign bit) is 1
                    imm -= (1 << 13)  # Apply sign extension by subtracting 2^13
                if InstructionInfo['func3'] == '000':
                    if self.myRF.readRF(rs1) == self.myRF.readRF(rs2):
                        self.PC = self.PC + int(imm / 4) - 1
                elif InstructionInfo['func3'] == '001':
                    if self.myRF.readRF(rs1) != self.myRF.readRF(rs2):
                        self.PC = self.PC + int(imm / 4) - 1

            elif InstructionInfo['opcode'] == op_U:
                rd = int(InstructionInfo['rd'], 2)
                imm = int(InstructionInfo['imm[31:12]'], 2)

            elif InstructionInfo['opcode'] == op_J: # JAL
                rd = int(InstructionInfo['rd'], 2)
                imm_binary = (InstructionInfo['imm[20,10:1,11,19:12]'][0] + InstructionInfo['imm[20,10:1,11,19:12]'][-8:] +
                              InstructionInfo['imm[20,10:1,11,19:12]'][11] + InstructionInfo['imm[20,10:1,11,19:12]'][1:11] + '0')
                imm = int(imm_binary, 2)
                # Sign extension for 21-bit immediate (if the 21th bit is 1, the value is negative)
                if imm_binary[0] == '1':  # Check if the highest bit (sign bit) is 1
                    imm -= (1 << 21)  # Apply sign extension by subtracting 2^21
                self.PC = self.PC + int(imm / 4) - 1
```

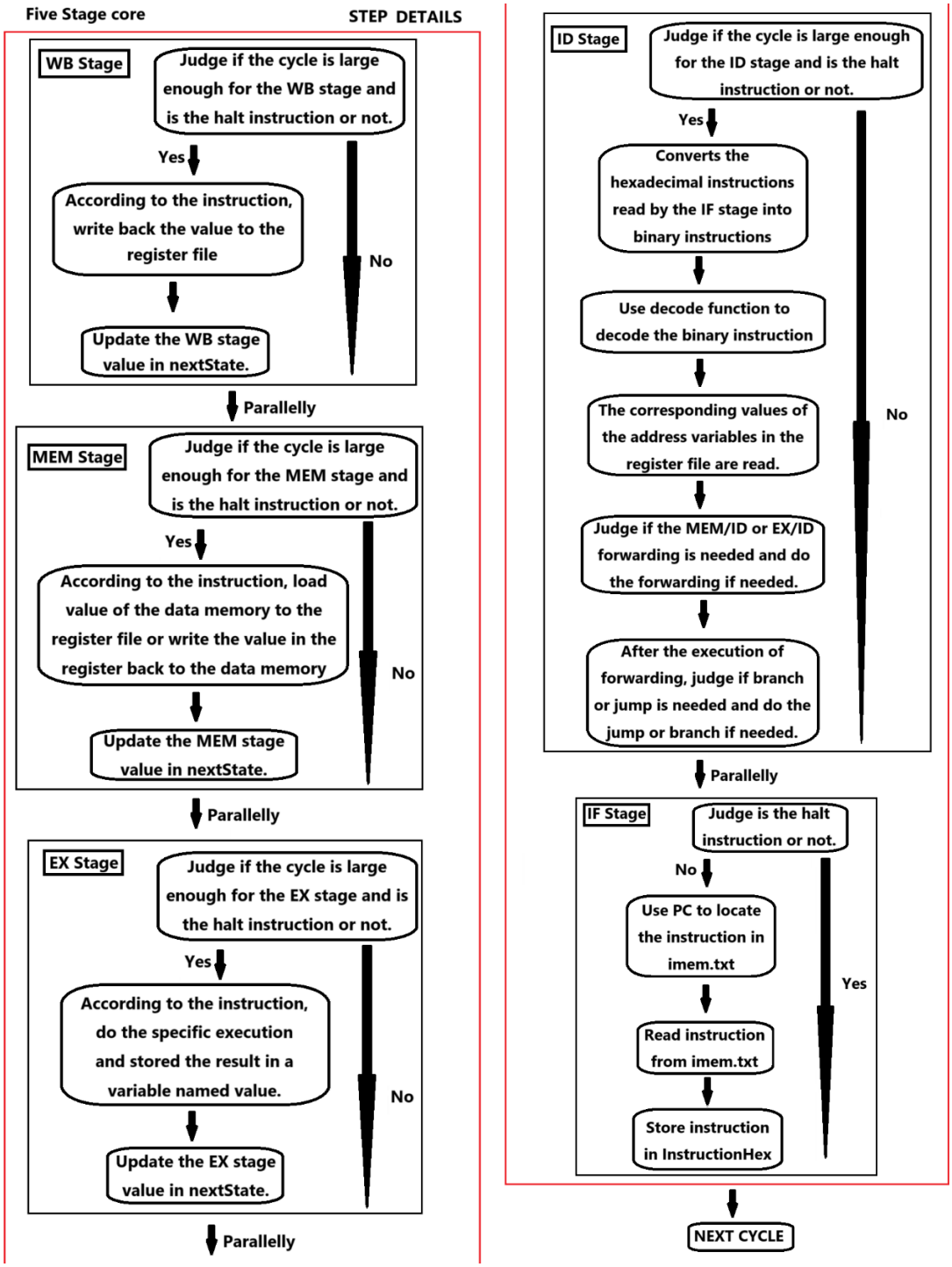
(continuing next page)

```
        #MEM stage
        if self.state.IF["nop"] == False:
            if InstructionInfo['opcode'] in op_R:
                if InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '000':  # ADD
                    value = int(self.myRF.readRF(rs1)) + int(self.myRF.readRF(rs2))
                elif InstructionInfo['func7'] == '0100000' and InstructionInfo['func3'] == '000':  # SUB
                    value = int(self.myRF.readRF(rs1)) - int(self.myRF.readRF(rs2))
                elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '100':  # XOR
                    value = int(self.myRF.readRF(rs1)) ^ int(self.myRF.readRF(rs2))
                elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '110':  # OR
                    value = int(self.myRF.readRF(rs1)) | int(self.myRF.readRF(rs2))
                elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '111':  # AND
                    value = int(self.myRF.readRF(rs1)) & int(self.myRF.readRF(rs2))

            elif InstructionInfo['opcode'] in op_I:
                if InstructionInfo['opcode'] == '0000011':  # LW
                    value = self.ext_dmem.readDataMem(rs1 + imm)
                elif InstructionInfo['opcode'] == '0010011':
                    if InstructionInfo['func3'] == '000':  # ADDI
                        value = int(self.myRF.readRF(rs1)) + imm
                    elif InstructionInfo['func3'] == '100':  # XORI
                        value = int(self.myRF.readRF(rs1)) ^ imm
                    elif InstructionInfo['func3'] == '110':  # ORI
                        value = int(self.myRF.readRF(rs1)) | imm
                    elif InstructionInfo['func3'] == '111':  # ANDI
                        value = int(self.myRF.readRF(rs1)) & imm

            elif InstructionInfo['opcode'] == op_S:  # SW
                value = self.myRF.readRF(rs2)

            elif InstructionInfo['opcode'] == op_B: #BEQ & BNE
                pass

            elif InstructionInfo['opcode'] == op_U:
                pass

            elif InstructionInfo['opcode'] == op_J: # JAL
                value = (self.PC - int(imm / 4) + 1)*4 + 4
        #WB stage
        if self.state.IF["nop"] == False:
            if InstructionInfo['opcode'] in op_R or InstructionInfo['opcode'] in op_I:
                self.myRF.writeRF(rd, value)

            elif InstructionInfo['opcode'] == op_S:  # SW
                self.ext_dmem.writeDataMem(address, value)

            elif InstructionInfo['opcode'] == op_B: # BNE & BEQ
                pass

            elif InstructionInfo['opcode'] == op_U:
                pass

            elif InstructionInfo['opcode'] == op_J: # JAL
                self.myRF.writeRF(rd, value)
            self.instructionCount += 1
        # ---------------------------------------------------------------------------------
        self.myRF.outputRF(self.cycle) # dump RF
        if not self.state.IF["nop"]:
            self.PC += 1
        self.nextState.IF["PC"] = self.PC*4
        print("\n")
        self.printState(self.nextState, self.cycle) # print states after executing cycle 0, cycle 1, cycle 2 ...
        self.state = self.nextState #The end of the cycle and updates the current state with the values calculated in this cycle
        self.cycle += 1

        if self.state.IF["nop"] and self.halted:
            print("----------------------------Single Stage Core Performance Metrics----------------------------\n")
            print("Number of cycles taken: " + str(self.cycle) + "\n")
            print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
            print("Cycles per instruction: " + str(self.cycle / self.instructionCount) + "\n")
            print("Instructions per cycle: " + str(self.instructionCount / self.cycle) + "\n")


    def printState(self, state, cycle):  1 个用法
        printstate = ["-"*70+"\n", "State after executing cycle: " + str(cycle) + "\n"]
        printstate.append("IF.PC: " + str(state.IF["PC"]) + "\n")
```

Task 2:

Five Stage:

The schematic for a five stages processor is as following:



The detailed code in my implementation is as follows.

class InsMem: it is used to access instructions from imem.txt

```
class InsMem(object):  1 个用法
    def __init__(self, name, ioDir):
        self.id = name
        with open(ioDir + "\\imem.txt") as im:
            self.IMem = [data.replace( _old: "\n",  _new: "") for data in im.readlines()]

    def readInstr(self, ReadAddress):  2 用法
        #read instruction memory
        #return 32 bit hex val
        indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
        selected_strings = [self.IMem[i] for i in indices_to_select]
        for i in range(4):
            selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
        instruction = ''.join(selected_strings)
        return instruction

```

class DataMem: it is used to access and write back data memory.

```
class DataMem(object):  2 用法
    def __init__(self, name, ioDir):
        self.id = name
        self.ioDir = ioDir
        self.DMem = ["00000000"]*MemSize
        with open(ioDir + "\\dmem.txt") as dm:
            self.DMem = [data.replace( _old: "\n",  _new: "") for data in dm.readlines()]
        self.DMem.extend(["00000000"] * (MemSize - len(self.DMem)))

    def readDataMem(self, ReadAddress):  2 个用法 (2 个动态)
        #read data memory
        #return 32 bit hex val
        indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
        selected_strings = [self.DMem[i] for i in indices_to_select]
        #for i in range(4):
        #    selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
        DataMem = ''.join(selected_strings)
        is_negative = DataMem[0] == '1'
        if is_negative:
            inverted_binary = ''.join('1' if b == '0' else '0' for b in DataMem)  # 取反
            positive_value = int(inverted_binary, 2) + 1  # 加1，得到正数值
            DataMem = -positive_value  # 加符号
        else:
            DataMem = int(DataMem, 2)
        print("data in mem is "+str(DataMem))
        return DataMem

    def writeDataMem(self, Address, WriteData):  2 个用法 (2 个动态)
        # write data into byte addressable memory
        if WriteData < 0:
            binary_str = bin((1 << 32) + WriteData)[2:]  # 加 2^32 实现补码
        else:
            binary_str = bin(WriteData)[2:]
        binaryData = binary_str.zfill(32)
        binary_8bit_Data = [binaryData[i:i + 8] for i in range(0, 32, 8)]
        required_lines = Address + 4
        while len(self.DMem) < required_lines:
            self.DMem.append("\n")
        for i in range(4):
            self.DMem[Address + i] = binary_8bit_Data[i]

    def outputDataMem(self):  2 用法
        resPath = self.ioDir + "\\" + self.id + "_DMEMResult.txt"
        print(self.DMem)
        with open(resPath, "w") as rp:
            i = 0
            for data in self.DMem:
                if data != "\n" and i != len(self.DMem)-1:
                    rp.writelines(str(data)+"\n")
                else:
                    rp.writelines(str(data))
                i += 1

class RegisterFile(object):  1 个用法
```

class RegisterFile: it is used to read and modify the registers' values.

```python
class RegisterFile(object):  1 个用法
    def __init__(self, ioDir):
        self.outputFile = ioDir + "RFResult.txt"
        binary_str = format(0, '032b')
        self.Registers = [binary_str for i in range(32)]

    def readRF(self, Reg_addr):  21 用法
        DataReg = self.Registers[Reg_addr]

        is_negative = DataReg[0] == '1'
        if is_negative:
            inverted_binary = ''.join('1' if b == '0' else '0' for b in DataReg)  # 取反
            positive_value = int(inverted_binary, 2) + 1  # 加1，得到正数值
            DataReg = -positive_value  # 加符号
        else:
            DataReg = int(DataReg, 2)

        return DataReg

    def writeRF(self, Reg_addr, Wrt_reg_data):  4 用法
        if Wrt_reg_data < 0:
            binary_str = bin((1 << 32) + Wrt_reg_data)[2:]  # 加 2^32 实现补码
        else:
            binary_str = bin(Wrt_reg_data)[2:]
        binaryData = binary_str.zfill(32)
        self.Registers[Reg_addr] = binaryData

    def outputRF(self, cycle):  2 用法
        op = ["-"*70+"\n", "State of RF after executing cycle:" + str(cycle) + "\n"]
        op.extend([str(val)+"\n" for val in self.Registers])
        if(cycle == 0): perm = "w"
        else: perm = "a"
        with open(self.outputFile, perm) as file:
            file.writelines(op)
```

Write back stage uses a WriteBack function and use the logic bellow to realize the write back stage and update of the self.nextState. Following are the implementation of the Write back function and logic:

WriteBack function:

```python
def WriteBack(InstructionInfo, RegAddr, value):
    rd = RegAddr['rd']
    if self.halted == False:
        if InstructionInfo['opcode'] in op_R or InstructionInfo['opcode'] in op_I:
            self.myRF.writeRF(rd, value)

        elif InstructionInfo['opcode'] == op_S:  # SW
            pass
        elif InstructionInfo['opcode'] == op_B:  # BNE & BEQ
            pass
        elif InstructionInfo['opcode'] == op_U:
            pass
        elif InstructionInfo['opcode'] == op_J: # JAL
            self.myRF.writeRF(rd, value)
```

Logic:

```python
# ----------WB Stage----------
if self.cycle > 4 and self.InstructionHexSet[str(self.WBIndex-1)] == 'ffffffff':
    self.WBstop = True
    self.state.WB["nop"] = True
    self.nextState.WB["nop"] = True
    print("WB STOP!")
elif self.cycle > 3 and not self.state.WB["nop"] and not self.WBstop:
    WBIndex = str(self.WBIndex)
    print("WB STAGE: " + WBIndex)
    stop = False
    if self.InstructionHexSet[str(self.WBIndex)] == 'ffffffff':
        stop = True
    if not stop:
        WriteBack(self.InstructionSet[WBIndex], self.RegAddrSet[WBIndex], self.ValueSet[WBIndex])
        self.nextState.WB["Wrt_data"] = self.ValueSet[WBIndex]# Update WB state
        self.nextState.WB["Rs"] = self.RegAddrSet[WBIndex]['Rs']
        self.nextState.WB["Rt"] = self.RegAddrSet[WBIndex]['Rt']
        self.nextState.WB["Wrt_reg_addr"] = self.RegAddrSet[WBIndex]['rd']
        self.nextState.WB["wrt_enable"] = 1

    self.WBIndex = self.WBIndex+1
elif self.state.WB["nop"] and not self.WBstop:
    self.nextState.WB["nop"] = False
```

Memory access stage uses MemoryAccess and updateMEMNextState functions and use the logic bellow to realize the memory access stage and update of the self.nextState. Following are the implementation of the Memory access functions and logic:

MemoryAccess function:

```
451         def MemoryAccess(InstructionInfo, RegAddr, value):
452             rs1 = RegAddr['rs1']
453             imm = RegAddr['imm']
454             if self.halted == False:
455                 if InstructionInfo['opcode'] in op_R:
456                     pass
457                 elif InstructionInfo['opcode'] in op_I:
458                     if InstructionInfo['opcode'] == '0000011':  # LW
459                         address = value
460                         value = self.ext_dmem.readDataMem(address)
461 
462                 elif InstructionInfo['opcode'] == op_S:  # SW
463                     print("rs1 is "+str(rs1))
464                     print("address "+str((int(rs1)+int(imm)))+" value "+str(value))
465                     self.ext_dmem.writeDataMem((int(rs1)+int(imm)), value)
466 
467                 elif InstructionInfo['opcode'] == op_B: # BEQ & BNE
468                     pass
469                 elif InstructionInfo['opcode'] == op_U:
470                     pass
471                 elif InstructionInfo['opcode'] == op_J: # JAL
472                     pass
473             return value
```

updateMEMNextState function:

```
474         def updateMEMNextState():
475             if self.InstructionSet[str(self.MEMIndex)]['opcode'] in op_R:
476                 self.nextState.MEM["ALUresult"] = self.ValueSet[str(self.MEMIndex)]
477                 self.nextState.MEM["Rs"] = self.RegAddrSet[str(self.MEMIndex)]['Rs']
478                 self.nextState.MEM["Rt"] = self.RegAddrSet[str(self.MEMIndex)]['Rt']
479                 self.nextState.MEM["rd_mem"] = 0
480                 self.nextState.MEM["wrt_mem"] = 0
481                 self.nextState.MEM["wrt_enable"] = 1
482                 self.nextState.MEM["Wrt_reg_addr"] = self.RegAddrSet[str(self.MEMIndex)]['rd']
483                 self.nextState.MEM["Store_data"] = 0
484 
485             elif self.InstructionSet[str(self.MEMIndex)]['opcode'] in op_I:
486                 self.nextState.MEM["ALUresult"] = self.ValueSet[str(self.MEMIndex)]
487                 self.nextState.MEM["Rs"] = self.RegAddrSet[str(self.MEMIndex)]['Rs']
488                 self.nextState.MEM["Rt"] = 0
489                 self.nextState.MEM["rd_mem"] = 0
490                 self.nextState.MEM["wrt_mem"] = 0
491                 self.nextState.MEM["wrt_enable"] = 1
492                 self.nextState.MEM["Wrt_reg_addr"] = self.RegAddrSet[str(self.MEMIndex)]['rd']
493                 self.nextState.MEM["Store_data"] = 0
494 
495             elif self.InstructionSet[str(self.MEMIndex)]['opcode'] in op_S:
496                 self.nextState.MEM["ALUresult"] = 0
497                 self.nextState.MEM["Rs"] = self.RegAddrSet[str(self.MEMIndex)]['Rs']
498                 self.nextState.MEM["Rt"] = self.RegAddrSet[str(self.MEMIndex)]['Rt']
499                 self.nextState.MEM["rd_mem"] = 0
500                 self.nextState.MEM["wrt_mem"] = 1
501                 self.nextState.MEM["Store_data"] = self.ValueSet[str(self.MEMIndex)]
502                 self.nextState.MEM["wrt_enable"] = 0
503                 self.nextState.MEM["Wrt_reg_addr"] = 0
504 
505             elif self.InstructionSet[str(self.MEMIndex)]['opcode'] in op_B:
506                 self.nextState.MEM["ALUresult"] = 0
507                 self.nextState.MEM["Rs"] = self.RegAddrSet[str(self.MEMIndex)]['Rs']
508                 self.nextState.MEM["Rt"] = self.RegAddrSet[str(self.MEMIndex)]['Rt']
509                 self.nextState.MEM["rd_mem"] = 0
510                 self.nextState.MEM["wrt_mem"] = 0
511                 self.nextState.MEM["Store_data"] = 0
512                 self.nextState.MEM["wrt_enable"] = 0
513                 self.nextState.MEM["Wrt_reg_addr"] = 0
514 
515             elif self.InstructionSet[str(self.MEMIndex)]['opcode'] in op_J:
516                 self.nextState.MEM["ALUresult"] = self.ValueSet[str(self.MEMIndex)]
517                 self.nextState.MEM["Rs"] = self.RegAddrSet[str(self.MEMIndex)]['Rs']
518                 self.nextState.MEM["Rt"] = self.RegAddrSet[str(self.MEMIndex)]['Rt']
519                 self.nextState.MEM["rd_mem"] = 0
520                 self.nextState.MEM["wrt_mem"] = 0
521                 self.nextState.MEM["Store_data"] = 0
522                 self.nextState.MEM["wrt_enable"] = 1
523                 self.nextState.MEM["Wrt_reg_addr"] = self.RegAddrSet[str(self.MEMIndex)]['rd']
524 
```

Logic:

```
        # -----------MEM Stage-----------------------------------------------------------
        if self.cycle > 3 and self.InstructionHexSet[str(self.MEMIndex-1)] == 'ffffffff':
            self.MEMstop = True
            self.state.MEM["nop"] = True
            self.nextState.MEM["nop"] = True
            print("MEM STOP!")

        if self.cycle > 2 and not self.state.MEM["nop"] and not self.MEMstop:
            MEMIndex = str(self.MEMIndex)
            print("MEM STAGE: " + MEMIndex)
            stop = False
            if self.InstructionHexSet[str(self.MEMIndex)] == 'ffffffff':
                stop = True
            if not stop:#self.InstructionHexSet[str(self.IFIndex-1)] != 'ffffffff':
                self.ValueSet[MEMIndex] = MemoryAccess(self.InstructionSet[MEMIndex], self.RegAddrSet[MEMIndex],
                                                       self.ValueSet[MEMIndex])
                updateMEMNextState()
            print("after MEM, value is "+str(self.nextState.MEM["ALUresult"]))
            self.MEMIndex = self.MEMIndex + 1

        elif self.state.MEM["nop"] and not self.MEMstop:
            self.nextState.MEM["nop"] = False
            self.nextState.WB["nop"] = True
```

Execution stage uses Execution and updateEXNextState functions and use the logic bellow to realize the execution stage and update of the self.nextState. Following are the implementation of the Execution functions and logic:

Execution:

```
        def Execution(InstructionInfo, RegAddr):
            value = None
            if InstructionInfo is not None:
                rs1 = RegAddr['rs1']
                rs2 = RegAddr['rs2']
                imm = RegAddr['imm']
                if self.halted == False:
                    if InstructionInfo['opcode'] in op_R:
                        if InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '000':  # ADD
                            value = int(rs1) + int(rs2)
                        elif InstructionInfo['func7'] == '0100000' and InstructionInfo['func3'] == '000':  # SUB
                            value = int(rs1) - int(rs2)
                        elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '100':  # XOR
                            value = int(rs1) ^ int(rs2)
                        elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '110':  # OR
                            value = int(rs1) | int(rs2)
                        elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '111':  # AND
                            value = int(rs1) & int(rs2)

                    elif InstructionInfo['opcode'] in op_I:
                        if InstructionInfo['opcode'] == '0000011':  # LW
                            value = int(rs1) + int(imm)
                        elif InstructionInfo['opcode'] == '0010011':
                            if InstructionInfo['func3'] == '000':  # ADDI
                                value = int(rs1) + int(imm)
                                print("R4 IS VALUE OF: "+str(value))
                            elif InstructionInfo['func3'] == '100':  # XORI
                                value = int(rs1) ^ int(imm)
                            elif InstructionInfo['func3'] == '110':  # ORI
                                value = int(rs1) | int(imm)
                            elif InstructionInfo['func3'] == '111':  # ANDI
                                value = int(rs1) & int(imm)

                    elif InstructionInfo['opcode'] == op_S:  # SW
                        value = int(rs2)

                    elif InstructionInfo['opcode'] == op_B:  # BEQ & BNE
                        value = None

                    elif InstructionInfo['opcode'] == op_U:
                        pass

                    elif InstructionInfo['opcode'] == op_J:  # JAL
                        value = (self.JALPC-1) * 4 + 4
                        print("Value of JAL IN rd is "+str(value))

                if InstructionInfo is None:
                    value = None
            return value
```

updateEXNextState:

```
        def updateEXNextState():
            if self.InstructionSet[str(self.EXIndex)] is not None:
                if self.InstructionSet[str(self.EXIndex)]['opcode'] in op_R:
                    self.nextState.EX["Read_data1"] = self.RegAddrSet[str(self.EXIndex)]['rs1']
                    self.nextState.EX["Read_data2"] = self.RegAddrSet[str(self.EXIndex)]['rs2']
                    self.nextState.EX["Imm"] = self.RegAddrSet[str(self.EXIndex)]['imm']
                    self.nextState.EX["Rs"] = self.RegAddrSet[str(self.EXIndex)]['Rs']
                    self.nextState.EX["Rt"] = self.RegAddrSet[str(self.EXIndex)]['Rt']
                    self.nextState.EX["Wrt_reg_addr"] = self.RegAddrSet[str(self.EXIndex)]['rd']
                    self.nextState.EX["is_I_type"] = False
                    self.nextState.EX["wrt_enable"] = 1
                    #self.nextState.EX["alu_op"] = "10"
                    self.nextState.EX["alu_op"] = self.ValueSet[str(self.EXIndex)]
                    self.nextState.EX["rd_mem"] = 0
                    self.nextState.EX["wrt_mem"] = 0

                elif self.InstructionSet[str(self.EXIndex)]['opcode'] in op_I:
                    self.nextState.EX["Read_data1"] = self.RegAddrSet[str(self.EXIndex)]['rs1']
                    self.nextState.EX["Read_data2"] = 0
                    self.nextState.EX["Imm"] = self.RegAddrSet[str(self.EXIndex)]['imm']
                    self.nextState.EX["Rs"] = self.RegAddrSet[str(self.EXIndex)]['Rs']
                    self.nextState.EX["Rt"] = 0
                    self.nextState.EX["Wrt_reg_addr"] = self.RegAddrSet[str(self.EXIndex)]['rd']
                    self.nextState.EX["is_I_type"] = True
                    self.nextState.EX["wrt_enable"] = 1
                    # self.nextState.EX["alu_op"] = "10"
                    self.nextState.EX["alu_op"] = self.ValueSet[str(self.EXIndex)]
                    self.nextState.EX["rd_mem"] = 0
                    self.nextState.EX["wrt_mem"] = 0
                    if self.InstructionSet[str(self.EXIndex)]['opcode'] == '0000011':  # LW
                        self.nextState.EX["rd_mem"] = 1  # need to read mem
                        #self.nextState.EX["alu_op"] = "00"

                elif self.InstructionSet[str(self.EXIndex)]['opcode'] in op_S:
                    self.nextState.EX["Read_data1"] = self.RegAddrSet[str(self.EXIndex)]['rs1']
                    self.nextState.EX["Read_data2"] = self.RegAddrSet[str(self.EXIndex)]['rs2']
                    self.nextState.EX["Imm"] = self.RegAddrSet[str(self.EXIndex)]['imm']
                    self.nextState.EX["Rs"] = self.RegAddrSet[str(self.EXIndex)]['Rs']
                    self.nextState.EX["Rt"] = self.RegAddrSet[str(self.EXIndex)]['Rt']
                    self.nextState.EX["Wrt_reg_addr"] = 0
                    self.nextState.EX["is_I_type"] = False
                    # self.nextState.EX["alu_op"] = "00"
                    self.nextState.EX["alu_op"] = self.ValueSet[str(self.EXIndex)]
                    self.nextState.EX["wrt_enable"] = 0
                    self.nextState.EX["rd_mem"] = 0
                    self.nextState.EX["wrt_mem"] = 1  # need to write mem

                elif self.InstructionSet[str(self.EXIndex)]['opcode'] in op_B:
                    self.nextState.EX["Read_data1"] = self.RegAddrSet[str(self.EXIndex)]['rs1']
                    self.nextState.EX["Read_data2"] = self.RegAddrSet[str(self.EXIndex)]['rs2']
                    self.nextState.EX["Imm"] = self.RegAddrSet[str(self.EXIndex)]['imm']
                    self.nextState.EX["Rs"] = self.RegAddrSet[str(self.EXIndex)]['Rs']
                    self.nextState.EX["Rt"] = self.RegAddrSet[str(self.EXIndex)]['Rt']
                    self.nextState.EX["Wrt_reg_addr"] = 0
                    self.nextState.EX["is_I_type"] = False
                    # self.nextState.EX["alu_op"] = "01"
                    self.nextState.EX["alu_op"] = self.ValueSet[str(self.EXIndex)]
                    self.nextState.EX["wrt_enable"] = 0
                    self.nextState.EX["rd_mem"] = 0
                    self.nextState.EX["wrt_mem"] = 0

                elif self.InstructionSet[str(self.EXIndex)]['opcode'] in op_J:
                    self.nextState.EX["Read_data1"] = 0
                    self.nextState.EX["Read_data2"] = 0
                    self.nextState.EX["Imm"] = self.RegAddrSet[str(self.EXIndex)]['imm']
                    self.nextState.EX["Rs"] = 0
                    self.nextState.EX["Rt"] = 0
                    self.nextState.EX["Wrt_reg_addr"] = self.RegAddrSet[str(self.EXIndex)]['rd']
                    self.nextState.EX["is_I_type"] = False
                    # self.nextState.EX["alu_op"] = "10"
                    self.nextState.EX["alu_op"] = self.ValueSet[str(self.EXIndex)]
                    self.nextState.EX["wrt_enable"] = 1
                    self.nextState.EX["rd_mem"] = 0
                    self.nextState.EX["wrt_mem"] = 0
```

Logic:

```
        # -----------EX Stage-----------------------------------------------------
        if str(self.EXIndex-1) in self.InstructionSet and self.InstructionHexSet[str(self.EXIndex-1)] == 'ffffffff':
            self.EXstop = True
            self.state.EX["nop"] = True
            self.nextState.EX["nop"] = True
            print("EX STOP!")

        if self.cycle > 1 and not self.state.EX["nop"] and not self.EXstop:
            EXIndex = str(self.EXIndex)
            print("EX STAGE: " + EXIndex)
            stop = False
            if self.InstructionHexSet[str(self.EXIndex)] == 'ffffffff':
                stop = True
            if not stop:
                self.ValueSet[EXIndex] = Execution(self.InstructionSet[EXIndex], self.RegAddrSet[EXIndex])
                updateEXNextState()
            self.EXIndex = self.EXIndex + 1
            self.nextState.MEM["nop"] = False
        elif self.state.EX["nop"] and not self.EXstop:
            self.nextState.EX["nop"] = False
            self.nextState.MEM["nop"] = True
```

Instruction decode stage uses InstructionDecode, decode and RegRead functions and use the logic bellow to realize the execution stage. Following are the implementation of the Instruction decode functions and logic:

InstructionDecode:

```
        def InstructionDecode(Instructionh):
            InstructionBin = bin(int(Instructionh, 16))[2:].zfill(32)
            InstructionInfo = self.decode(InstructionBin)
            return InstructionInfo
```

decode:

```
    def decode(self, instruction):  1 个用法
        instructionInfo_R = {'func7': '', 'rs2': '', 'rs1': '', 'func3': '', 'rd': '', 'opcode': ''}  # type
        instructionInfo_I = {'imm[11:0]': '', 'rs1': '', 'func3': '', 'rd': '', 'opcode': ''}
        instructionInfo_S = {'imm[11:5]': '', 'rs2': '', 'rs1': '', 'func3': '', 'imm[4:0]': '', 'opcode': ''}
        instructionInfo_B = {'imm[12,10:5]': '', 'rs2': '', 'rs1': '', 'func3': '', 'imm[4:1,11]': '', 'opcode': ''}
        instructionInfo_J = {'imm[20,10:1,11,19:12]': '', 'rd': '', 'opcode': ''}
        instructionInfo = {}
        opcode = instruction[-7:]
        op_R = ('0110011', '0111011')
        op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
        op_S = '0100011'
        op_B = '1100011'
        op_U = '0110111'
        op_J = '1101111'
        if opcode in op_R:
            instructionInfo_R['func7'] = instruction[:7]
            instructionInfo_R['rs2'] = instruction[7:12]
            instructionInfo_R['rs1'] = instruction[12:17]
            instructionInfo_R['func3'] = instruction[17:20]
            instructionInfo_R['rd'] = instruction[20:25]
            instructionInfo_R['opcode'] = instruction[-7:]
            instructionInfo = instructionInfo_R
        elif opcode in op_I:
            instructionInfo_I['imm[11:0]'] = instruction[:12]
            instructionInfo_I['rs1'] = instruction[12:17]
            instructionInfo_I['func3'] = instruction[17:20]
            instructionInfo_I['rd'] = instruction[20:25]
            instructionInfo_I['opcode'] = instruction[-7:]
            instructionInfo = instructionInfo_I
        elif opcode == op_S:
            instructionInfo_S['imm[11:5]'] = instruction[:7]
            instructionInfo_S['rs2'] = instruction[7:12]
            instructionInfo_S['rs1'] = instruction[12:17]
            instructionInfo_S['func3'] = instruction[17:20]
            instructionInfo_S['imm[4:0]'] = instruction[20:25]
            instructionInfo_S['opcode'] = instruction[-7:]
            instructionInfo = instructionInfo_S
        elif opcode == op_B:
            instructionInfo_B['imm[12,10:5]'] = instruction[:7]
            instructionInfo_B['rs2'] = instruction[7:12]
            instructionInfo_B['rs1'] = instruction[12:17]
            instructionInfo_B['func3'] = instruction[17:20]
            instructionInfo_B['imm[4:1,11]'] = instruction[20:25]
            instructionInfo_B['opcode'] = instruction[-7:]
            instructionInfo = instructionInfo_B
        elif opcode == op_J:
            instructionInfo_J['imm[20,10:1,11,19:12]'] = instruction[:20]
            instructionInfo_J['rd'] = instruction[20:25]
            instructionInfo_J['opcode'] = instruction[-7:]
            instructionInfo = instructionInfo_J
        elif opcode == '1111111':
            instructionInfo = None
        else:
            print(f"something wrong with the instruction decoding!!!")
        return instructionInfo
```

RegRead:

```
    def RegRead(InstructionInfo):
        op_R = ('0110011', '0111011')
        op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
        op_S = '0100011'
        op_B = '1100011'
        op_J = '1101111'
        rs1 = 0
        rs2 = 0
        rd = 0
        imm = 0

        if InstructionInfo is None:
            Regread = None
        elif InstructionInfo is not None:
            if InstructionInfo['opcode'] in op_R:
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                rd = int(InstructionInfo['rd'], 2)

            elif InstructionInfo['opcode'] in op_I:
                rs1 = int(InstructionInfo['rs1'], 2)
                rd = int(InstructionInfo['rd'], 2)
                imm_str = InstructionInfo['imm[11:0]']
                if imm_str[0] == '1':
                    imm = int(imm_str, 2) - (1 << len(imm_str))
                else:
                    imm = int(imm_str, 2)

            elif InstructionInfo['opcode'] == op_S:
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                imm_str = InstructionInfo['imm[4:0]']
                imm = int(imm_str, 2)

            elif InstructionInfo['opcode'] == op_B:  # BEQ & BNE
                rs1 = int(InstructionInfo['rs1'], 2)
                rs2 = int(InstructionInfo['rs2'], 2)
                imm_binary = (InstructionInfo['imm[12,10:5]'][0] + InstructionInfo['imm[4:1,11]'][4] +
                              InstructionInfo['imm[12,10:5]'][1:7] + InstructionInfo['imm[4:1,11]'][:4] + '0')
                imm = int(imm_binary, 2)
                # Sign extension for 13-bit immediate (if the 13th bit is 1, the value is negative)
                if imm_binary[0] == '1':  # Check if the highest bit (sign bit) is 1
                    imm -= (1 << 13)  # Apply sign extension by subtracting 2^13
            elif InstructionInfo['opcode'] == op_J:  # JAL
                rd = int(InstructionInfo['rd'], 2)
                imm_binary = (InstructionInfo['imm[20,10:1,11,19:12]'][0] + InstructionInfo['imm[20,10:1,11,19:12]
                        InstructionInfo['imm[20,10:1,11,19:12]'][11] + InstructionInfo['imm[20,10:1,11,19:12]'][1:
                imm = int(imm_binary, 2)
                # Sign extension for 21-bit immediate (if the 21th bit is 1, the value is negative)
                if imm_binary[0] == '1':  # Check if the highest bit (sign bit) is 1
                    imm -= (1 << 21)  # Apply sign extension by subtracting 2^21
                self.JALPC = self.PC
                self.PC = self.PC + int(imm / 4) - 2

                self.state.IF["nop"] = True
                self.nextState.ID["nop"] = True
                print("stop IF here for JAL and next State ID stop!!!!!!!!!!!!!!!!!!!")

            Regread = {'rs1': self.myRF.readRF(rs1), 'rs2': self.myRF.readRF(rs2), 'rd': rd, 'imm': imm,
                       'Rs': rs1, 'Rt': rs2}
            # judge if forwarding for ID is needed and forwarding realization below--------------------------------
            def NOTBSType(x):
                if x in (op_B, op_S):
                    return False
                else:
                    return True

            if self.InstructionSet[str(self.IDIndex)]['opcode'] in (tuple(op_R) + (op_B, op_S)) and (not self.LUHS
                if self.IDIndex>0 and NOTBSType(self.InstructionSet[str(self.IDIndex-1)]['opcode']) and rs1 == sel
                    Regread['rs1'] = self.nextState.EX["alu_op"]
                    print("forwarding for rs1 from EX with value: "+str(Regread['rs1']))
                if self.IDIndex>1 and NOTBSType(self.InstructionSet[str(self.IDIndex-2)]['opcode']) and rs1 == sel
                    Regread['rs1'] = self.nextState.MEM["ALUresult"]
                    print("forwarding for rs1 from MEM with value: "+str(Regread['rs1']))
                if self.IDIndex>0 and NOTBSType(self.InstructionSet[str(self.IDIndex-1)]['opcode']) and rs2 == sel
                    Regread['rs2'] = self.nextState.EX["alu_op"]
                    print("forwarding for rs2 from EX with value: "+str(Regread['rs2']))
                if self.IDIndex>1 and NOTBSType(self.InstructionSet[str(self.IDIndex-2)]['opcode']) and rs2 == sel
                    Regread['rs2'] = self.nextState.MEM["ALUresult"]
                    print("forwarding for rs2 from MEM with value: "+str(Regread['rs2']))

            elif self.InstructionSet[str(self.IDIndex)]['opcode'] in op_I and (not self.LUHSyet):
                if self.IDIndex>0 and NOTBSType(self.InstructionSet[str(self.IDIndex-1)]['opcode']) and rs1 == sel
                    Regread['rs1'] = self.nextState.EX["alu_op"]
                    print("forwarding for rs1 from EX")
                if self.IDIndex>1 and NOTBSType(self.InstructionSet[str(self.IDIndex-2)]['opcode']) and rs1 == sel
                    Regread['rs1'] = self.nextState.MEM["ALUresult"]
                    print("forwarding for rs1 from MEM")

            elif self.InstructionSet[str(self.IDIndex)]['opcode'] in op_J:
                pass

            elif self.InstructionSet[str(self.IDIndex)]['opcode'] in (tuple(op_R) + (op_B, op_S)) and self.LUHSyet
                if self.IDIndex>0 and (self.InstructionSet[str(self.IDIndex-1)]['opcode']=='0000011') and rs1 == s
                    Regread['rs1'] = self.nextState.MEM["ALUresult"]
                    print("forwarding for rs1 from MEM with value: "+str(Regread['rs1']))
                if self.IDIndex>0 and (self.InstructionSet[str(self.IDIndex-1)]['opcode']=='0000011') and rs2 == s
                    Regread['rs2'] = self.nextState.MEM["ALUresult"]
                    print("forwarding for rs2 from MEM with value: "+str(Regread['rs2']))


            if InstructionInfo['opcode'] == op_B:  # BEQ & BNE branch after forwarding
                if InstructionInfo['func3'] == '000':
                    if Regread['rs1'] == Regread['rs2']:
                        self.PC = self.PC + int(imm / 4) - 2
                        self.state.IF["nop"] = True
                        self.nextState.ID["nop"] = True
                        print("stop IF here for BEQ and next State ID stop!!!!!!!!!!!!!!!!!!!!")
                elif InstructionInfo['func3'] == '001':
                    if Regread['rs1'] != Regread['rs2']:
                        self.PC = self.PC + int(imm / 4) - 2
                        self.state.IF["nop"] = True
                        self.nextState.ID["nop"] = True
                        print("stop IF here for BNE and next State ID stop!!!!!!!!!!!!!!!!!!!!")
            return Regread

    def InstructionFetch():...
```

Logic:

```
        # ----------ID Stage-------------------------------------------------------
        if self.cycle > 1 and self.InstructionHexSet[str(self.IDIndex-1)] == 'ffffffff':
            self.IDstop = True
            self.state.ID["nop"] = True
            self.nextState.ID["nop"] = True
            self.nextState.ID["Instr"] = self.state.ID["Instr"]
            print("ID STOP!")
        elif self.cycle > 0 and not self.state.ID["nop"] and not self.IDstop:
            IDIndex = str(self.IDIndex)
            if not self.state.ID["nop"]:
                print("ID STAGE: " + IDIndex)
                InstructionH = self.InstructionHexSet[IDIndex]
                self.InstructionSet[IDIndex] = InstructionDecode(InstructionH)
                if (self.IDIndex>0 and (self.InstructionSet[str(self.IDIndex-1)]['opcode']=='0000011') and
                        ('rd' in self.InstructionSet[str(self.IDIndex-1)]) and
                        (('rs1' in self.InstructionSet[IDIndex] and self.InstructionSet[IDIndex]['rs1'] == self.Instruct
                        ('rs2' in self.InstructionSet[IDIndex] and self.InstructionSet[IDIndex]['rs2'] == self.Instructi
                        (not self.LUHSyet)):# Load-Use Hazard Detection
                    print("Load Use Hazard!!!!!!!!!!!!!!!!!!!")
                    self.nextState.ID["nop"] = False
                    self.nextState.EX["nop"] = True
                    self.LUHSyet = True
                else:
                    self.RegAddrSet[IDIndex] = RegRead(self.InstructionSet[IDIndex])
                    self.LUHSyet = False
                    print(self.RegAddrSet[IDIndex])
                    if self.InstructionSet[str(self.IDIndex)] is None:
                        self.nextState.ID["Instr"] = self.state.ID["Instr"]
                    elif self.InstructionSet[str(self.IDIndex)] is not None:
                        self.nextState.ID["Instr"] = bin(int(self.InstructionHexSet[str(self.IDIndex)], 16))[2:].zfill(3
                    self.IDIndex = self.IDIndex + 1


        elif self.state.ID["nop"]:
            self.nextState.ID["nop"] = False
            self.nextState.EX["nop"] = True
```

Instruction fetch stage uses InstructionFetch function and use the logic bellow to realize the execution stage. Following are the implementation of the Instruction fetch function and logic:

InstructionFetch:

```
        def InstructionFetch():
            Instructionhex = imem.readInstr(self.PC*4)
            return Instructionhex
```

Logic:

```
        #----------IF Stage-------------------------------------------------------
        if self.cycle > 0 and self.InstructionHexSet[str(self.IFIndex-1)] == 'ffffffff':
            self.IFstop = True
            self.state.IF["nop"] = True
            self.nextState.IF["nop"] = True
            print("IF STOP!")
        elif not self.state.IF["nop"] and not self.IFstop:
            IFIndex = str(self.IFIndex)
            InstructionHex = InstructionFetch()
            self.InstructionHexSet[IFIndex] = InstructionHex
            print("IF STAGE: " + str(self.IFIndex) + " " + InstructionHex)
            self.IFIndex = self.IFIndex + 1
        if self.state.IF["nop"]:
            self.nextState.IF["nop"] = False
        self.nextState.IF["PC"] = (self.PC+1)*4
```

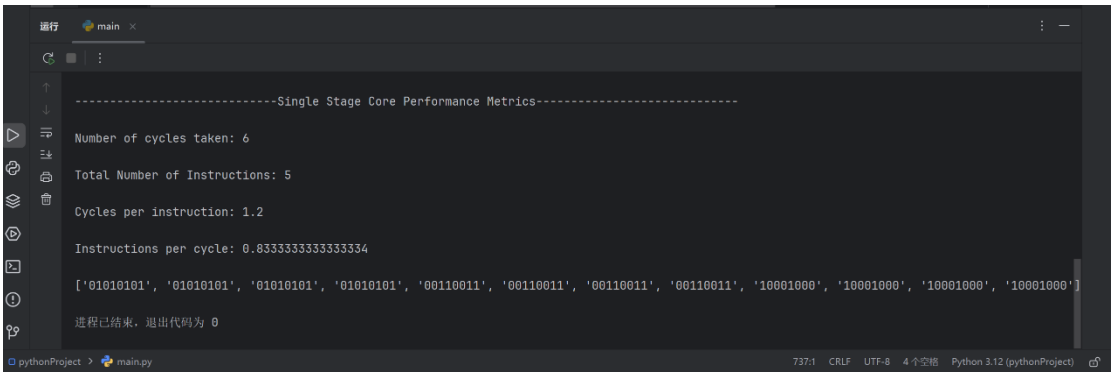
The class FiveStageCore(Core) is implemented as follows:

Task 3:

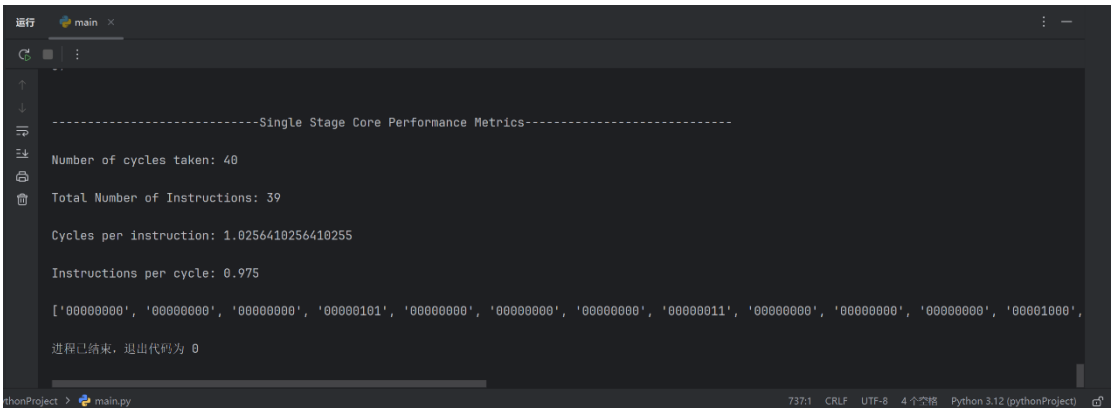
For SingleCore, use the following code in step() function to output the performance marix.

```
315         if self.state.IF["nop"] and self.halted:
316             print("-----------------------------Single Stage Core Performance Metrics-----------------------------\n")
317             print("Number of cycles taken: " + str(self.cycle) + "\n")
318             print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
319             print("Cycles per instruction: " + str(self.cycle / self.instructionCount) + "\n")
320             print("Instructions per cycle: " + str(self.instructionCount / self.cycle) + "\n")
```

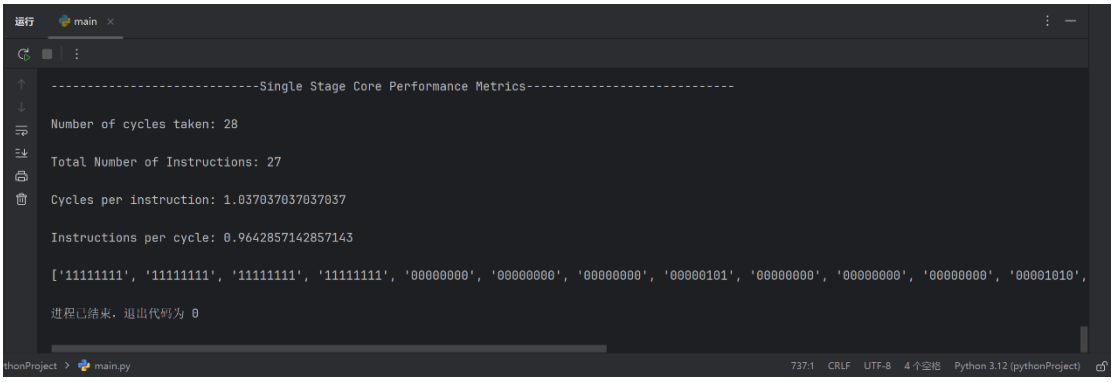
the output matrix for testcase0 is:



The output matrix for testcase1 is:



The output matrix for testcase2 is:



The output file (StateResult_SS, SS_RFResult, SS_DMEMResult) together with the above PerformanceMetrics_Result for testcase 0, 1 and 2 are packed up in submission folder. After comparison, all results are identical to the ones provided by brightspace, and you can run main.py for a second comparison.

For FiveStageCore, use the following code in step() function to output the performance marix.

```
if self.IFstop and self.IDstop and self.EXstop and self.MEMstop and self.WBstop:
    self.halted = True
    if self.InstructionSet[str(self.WBIndex-2)]['opcode'] == op_B:
        print("----------------------------Five Stage Core Performance Metrics----------------------------\n")
        print("Number of cycles taken: " + str(self.cycle-3) + "\n")
        print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
        print("Cycles per instruction: " + str((self.cycle-3) / self.instructionCount) + "\n")
        print("Instructions per cycle: " + str(self.instructionCount / (self.cycle-3)) + "\n")
    else:
        print("----------------------------Five Stage Core Performance Metrics----------------------------\n")
        print("Number of cycles taken: " + str(self.cycle) + "\n")
        print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
        print("Cycles per instruction: " + str((self.cycle) / self.instructionCount) + "\n")
        print("Instructions per cycle: " + str(self.instructionCount / (self.cycle)) + "\n")
```

the output matrix for testcase0 is:

```
----------------------------Five Stage Core Performance Metrics----------------------------

Number of cycles taken: 10

Total Number of Instructions: 5

Cycles per instruction: 2.0

Instructions per cycle: 0.5
```

The output matrix for testcase1 is:

```
----------------------------Five Stage Core Performance Metrics----------------------------

Number of cycles taken: 46

Total Number of Instructions: 39

Cycles per instruction: 1.1794871794871795

Instructions per cycle: 0.8478260869565217
```

The output matrix for testcase2 is:

```
----------------------------Five Stage Core Performance Metrics----------------------------

Number of cycles taken: 38

Total Number of Instructions: 27

Cycles per instruction: 1.4074074074074074

Instructions per cycle: 0.7105263157894737
```

The output file (StateResult_SS, SS_RFResult, SS_DMEMResult) together with the above PerformanceMetrics_Result for testcase 0, 1 and 2 are packed up in submission folder. After comparison, all results are identical to the ones provided by brightspace, and you can run main.py for a second comparison.

Task 4:

As the results shown above, the number of cycles of SingleCore is always one more than the number of instructions due to the HALT delay. As the number of instructions approaches to infinity, the value of 'CPI' and 'IPC' approaches to 1.

In addition, the cycles taken by FiveStageCore is always more than SingleCore. However, this is because the stages are divided into five parallel stages. In addition, the cycle time of FiveStageCore is shorter than the cycle time of SingleCore.

The execution time of the program is calculated as:

$$ET = number\ of\ cycles \times cycle\ time$$

So that for testcase 0, if the cycle time of SingleCore is $\frac{10}{6} \approx 1.67$ times longer than the cycle time of FiveStageCore, the performance of FiveStageCore is better than SingleCore. Otherwise, the performance of SingleCore is better than FiveStageCore.

For testcase 1, if the cycle time of SingleCore is $\frac{46}{40} = 1.15$ times longer than the cycle time of FiveStageCore, the performance of FiveStageCore is better than SingleCore. Otherwise, the performance of SingleCore is better than FiveStageCore.

For testcase 2, if the cycle time of SingleCore is $\frac{38}{28} \approx 1.36$ times longer than the cycle time of FiveStageCore, the performance of FiveStageCore is better than SingleCore. Otherwise, the performance of SingleCore is better than FiveStageCore.

Task 5:

Optimizations or features can be added to improve performance:

1. Implement **dynamic branch prediction** or **loop unrolling** to reduce control hazards.
2. Add **fine-grained performance monitoring** and **visualizations** for better insights.
3. Introduce **instruction/data caches** and optimize memory access for latency reduction.
4. **Expand forwarding paths** and handle **multicycle operations**.