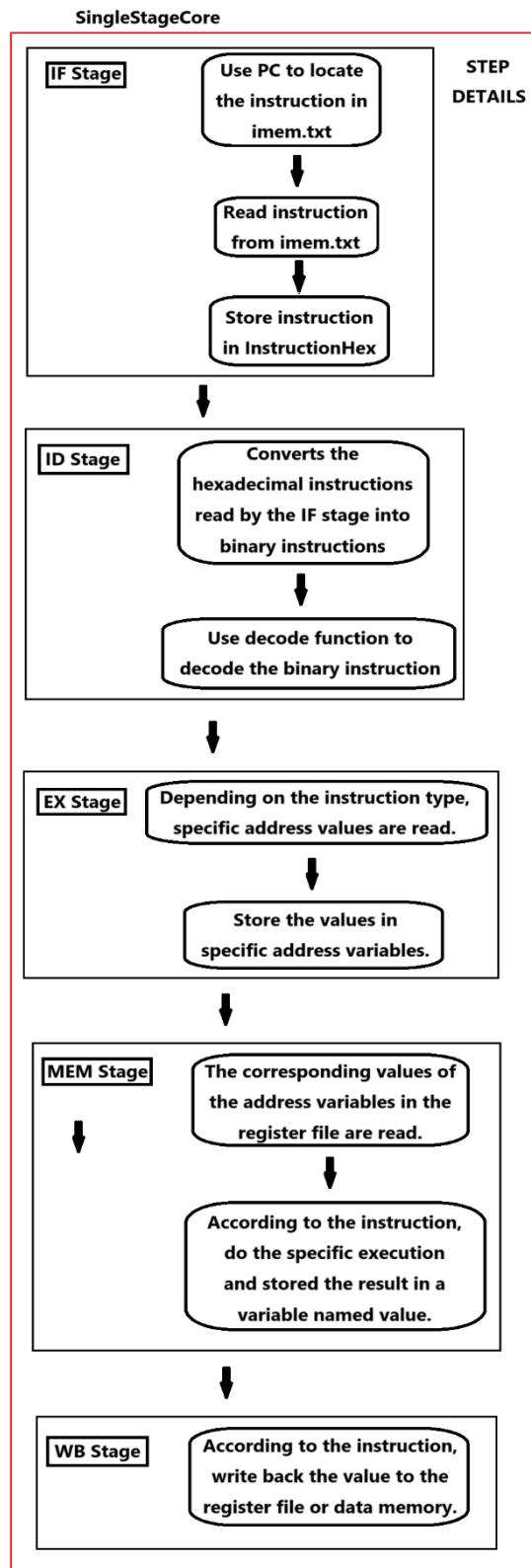


Phase 2 report

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

```
6     class InsMem(object): 1个用法
7         def __init__(self, name, ioDir):
8             self.id = name
9             with open(ioDir + "\\imem.txt") as im:
10                 self.IMem = [data.replace(_old: "\n", _new: "") for data in im.readlines()]
11
12         def readInstr(self, ReadAddress): 2用法
13             #read instruction memory
14             #return 32 bit hex val
15             indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
16             selected_strings = [self.IMem[i] for i in indices_to_select]
17             for i in range(4):
18                 selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
19             instruction = ''.join(selected_strings)
20             return instruction
21
```

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

```
22     class DataMem(object): 2用法
23         def __init__(self, name, ioDir):
24             self.id = name
25             self.ioDir = ioDir
26             self.DMem = ["00000000"]*MemSize
27             with open(ioDir + "\\dmem.txt") as dm:
28                 self.DMem = [data.replace(_old: "\n", _new: "") for data in dm.readlines()]
29             self.DMem.extend(["00000000"] * (MemSize - len(self.DMem)))
30
31         def readDataMem(self, ReadAddress): 2个用法(2个动态)
32             #read data memory
33             #return 32 bit hex val
34             indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
35             selected_strings = [self.DMem[i] for i in indices_to_select]
36             for i in range(4):
37                 selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
38             DataMem = ''.join(selected_strings)
39             is_negative = DataMem[0] == '1'
40             if is_negative:
41                 inverted_binary = ''.join('1' if b == '0' else '0' for b in DataMem) # 取反
42                 positive_value = int(inverted_binary, 2) + 1 # 加1, 得到正数值
43                 DataMem = -positive_value # 加符号
44             else:
45                 DataMem = int(DataMem, 2)
46             print("data in mem is "+str(DataMem))
47             return DataMem
48
49         def writeDataMem(self, Address, WriteData): 2个用法(2个动态)
50             # write data into byte addressable memory
51             if WriteData < 0:
52                 binary_str = bin((1 << 32) + WriteData)[2:] # 加 2^32 实现补码
53             else:
54                 binary_str = bin(WriteData)[2:]
55             binaryData = binary_str.zfill(32)
56             binary_8bit_Data = [binaryData[i:i + 8] for i in range(0, 32, 8)]
57             required_lines = Address + 4
58             while len(self.DMem) < required_lines:
59                 self.DMem.append("\n")
60             for i in range(4):
61                 self.DMem[Address + i] = binary_8bit_Data[i]
62
63         def outputDataMem(self): 2用法
64             resPath = self.ioDir + "\\" + self.id + "_DMEMResult.txt"
65             print(self.DMem)
66             with open(resPath, "w") as rp:
67                 i = 0
68                 for data in self.DMem:
69                     if data != "\n" and i != len(self.DMem)-1:
70                         rp.writelines(str(data)+"\n")
71                     else:
72                         rp.writelines(str(data))
73                     i += 1
74
75     class RegisterFile(object): 1个用法
```

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

```
75     class RegisterFile(object): 1个用法
76         def __init__(self, ioDir):
77             self.outputfile = ioDir + "RResult.txt"
78             binary_str = format(0, '032b')
79             self.Registers = [binary_str for i in range(32)]
80
81         def readRF(self, Reg_addr): 21用法
82             DataReg = self.Registers[Reg_addr]
83
84             is_negative = DataReg[0] == '1'
85             if is_negative:
86                 inverted_binary = ''.join('1' if b == '0' else '0' for b in DataReg) # 取反
87                 positive_value = int(inverted_binary, 2) + 1 # [01. 得到正数值
88                 DataReg = '-positive_value' # 加符号
89             else:
90                 DataReg = int(DataReg, 2)
91
92             return DataReg
93
94         def writeRF(self, Reg_addr, Wrt_reg_data): 4用法
95             if Wrt_reg_data < 0:
96                 binary_str = bin((1 << 32) + Wrt_reg_data)[2:] # 加 2^32 实现补码
97             else:
98                 binary_str = bin(Wrt_reg_data)[2:]
99             binaryData = binary_str.zfill(32)
100            self.Registers[Reg_addr] = binaryData
101
102        def outputRF(self, cycle): 2用法
103            op = ["--*78*\n", "State of RF after executing cycle:" + str(cycle) + "\n"]
104            op.append([str(val)+"\n" for val in self.Registers])
105            if(cycle == 0): perm = "w"
106            else: perm = "a"
107            with open(self.outputfile, perm) as file:
108                file.writelines(op)
```

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

```
118         def decode(self, instruction): 1个用法
119             instructionInfo = {}
120             opcode = instruction[-7:]
121             op_R = ('0110011', '0111011')
122             op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
123             op_S = '0100011'
124             op_B = '1100011'
125             op_U = '0110111'
126             op_J = '1101111'
127             if opcode in op_R:
128                 self.instructionInfo_R['func7'] = instruction[:7]
129                 self.instructionInfo_R['rs2'] = instruction[7:12]
130                 self.instructionInfo_R['rs1'] = instruction[12:17]
131                 self.instructionInfo_R['func3'] = instruction[17:20]
132                 self.instructionInfo_R['rd'] = instruction[20:25]
133                 self.instructionInfo_R['opcode'] = instruction[-7:]
134                 instructionInfo = self.instructionInfo_R
135             elif opcode in op_I:
136                 self.instructionInfo_I['imm[11:0]'] = instruction[:12]
137                 self.instructionInfo_I['rs1'] = instruction[12:17]
138                 self.instructionInfo_I['func3'] = instruction[17:20]
139                 self.instructionInfo_I['rd'] = instruction[20:25]
140                 self.instructionInfo_I['opcode'] = instruction[-7:]
141                 instructionInfo = self.instructionInfo_I
142             elif opcode == op_S:
143                 self.instructionInfo_S['imm[11:5]'] = instruction[:7]
144                 self.instructionInfo_S['rs2'] = instruction[7:12]
145                 self.instructionInfo_S['rs1'] = instruction[12:17]
146                 self.instructionInfo_S['func3'] = instruction[17:20]
147                 self.instructionInfo_S['imm[4:0]'] = instruction[20:25]
148                 self.instructionInfo_S['opcode'] = instruction[-7:]
149                 instructionInfo = self.instructionInfo_S
150             elif opcode == op_B:
151                 self.instructionInfo_B['imm[12,10:5]'] = instruction[:7]
152                 self.instructionInfo_B['rs2'] = instruction[7:12]
153                 self.instructionInfo_B['rs1'] = instruction[12:17]
154                 self.instructionInfo_B['func3'] = instruction[17:20]
155                 self.instructionInfo_B['imm[4:1,11]'] = instruction[20:25]
156                 self.instructionInfo_B['opcode'] = instruction[-7:]
157                 instructionInfo = self.instructionInfo_B
158             elif opcode == op_U:
159                 self.instructionInfo_U['imm[31:12]'] = instruction[:20]
160                 self.instructionInfo_U['rd'] = instruction[20:25]
161                 self.instructionInfo_U['opcode'] = instruction[-7:]
162                 instructionInfo = self.instructionInfo_U
163             elif opcode == op_J:
164                 self.instructionInfo_J['imm[20,10:1,11,19:12]'] = instruction[:20]
165                 self.instructionInfo_J['rd'] = instruction[20:25]
166                 self.instructionInfo_J['opcode'] = instruction[-7:]
167                 instructionInfo = self.instructionInfo_J
168             elif opcode == '1111111':
169                 self.state.IF["nop"] = True
170             else:
171                 print(f"something wrong with the instruction decoding!!!")
172
173             print("SS_OPCODE: " + opcode)
174             return instructionInfo
```

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

```

176     def step(self): 1个用法
177         # Your implementation
178         print("this is cycle: " + str(self.cycle))
179         rs1 = 0
180         rs2 = 0
181         rd = 0
182         imm = 0
183         value = 0
184         address = 0
185         op_R = ('0110011', '0111011')
186         op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
187         op_S = '0100011'
188         op_B = '1100011'
189         op_U = '0110111'
190         op_J = '1101111'
191
192         #IF stage
193         if self.state.IF["nop"]:
194             self.halted = True
195             self.instructionCount += 1
196         if self.state.IF["nop"] == False:
197             InstructionHex = imem.readInstr(self.PC*4)
198         #ID stage
199         if self.state.IF["nop"] == False:
200             InstructionBin = bin(int(InstructionHex, 16))[2:].zfill(32)
201             InstructionInfo = self.decode(InstructionBin)
202         #EX stage
203         if self.state.IF["nop"] == False:
204             if InstructionInfo['opcode'] in op_R:
205                 rs1 = int(InstructionInfo['rs1'], 2)
206                 rs2 = int(InstructionInfo['rs2'], 2)
207                 rd = int(InstructionInfo['rd'], 2)
208
209             elif InstructionInfo['opcode'] in op_I:
210                 rs1 = int(InstructionInfo['rs1'], 2)
211                 rd = int(InstructionInfo['rd'], 2)
212                 imm = int(InstructionInfo['imm[11:0]'], 2)
213
214             elif InstructionInfo['opcode'] == op_S:
215                 rs1 = int(InstructionInfo['rs1'], 2)
216                 rs2 = int(InstructionInfo['rs2'], 2)
217                 imm = int(InstructionInfo['imm[4:0]'], 2)
218                 address = rs1 + imm
219
220             elif InstructionInfo['opcode'] == op_B:
221                 rs1 = int(InstructionInfo['rs1'], 2)
222                 rs2 = int(InstructionInfo['rs2'], 2)
223                 imm_binary = (InstructionInfo['imm[12,10:5]'][0] + InstructionInfo['imm[4:1,11]'][4] +
224                             InstructionInfo['imm[12,10:5]'][1:7] + InstructionInfo['imm[4:1,11]'][4:1] + '0')
225                 imm = int(imm_binary, 2)
226                 # Sign extension for 13-bit immediate (if the 13th bit is 1, the value is negative)
227                 if imm_binary[0] == '1': # Check if the highest bit (sign bit) is 1
228                     imm -= (1 << 13) # Apply sign extension by subtracting 2^13
229                 if InstructionInfo['func3'] == '000':
230                     if self.myRF.readRF(rs1) == self.myRF.readRF(rs2):
231                         self.PC = self.PC + int(imm / 4) - 1
232                     elif InstructionInfo['func3'] == '001':
233                         if self.myRF.readRF(rs1) != self.myRF.readRF(rs2):
234                             self.PC = self.PC + int(imm / 4) - 1
235
236             elif InstructionInfo['opcode'] == op_U:
237                 rd = int(InstructionInfo['rd'], 2)
238                 imm = int(InstructionInfo['imm[31:12]'], 2)
239
240             elif InstructionInfo['opcode'] == op_J: # JAL
241                 rd = int(InstructionInfo['rd'], 2)
242                 imm_binary = (InstructionInfo['imm[20,10:1,11,19:12]'][0] + InstructionInfo['imm[20,10:1,11,19:12]'][-8:] +
243                             InstructionInfo['imm[20,10:1,11,19:12]'][11] + InstructionInfo['imm[20,10:1,11,19:12]'][1:11] + '0')
244                 imm = int(imm_binary, 2)
245                 # Sign extension for 21-bit immediate (if the 21th bit is 1, the value is negative)
246                 if imm_binary[0] == '1': # Check if the highest bit (sign bit) is 1
247                     imm -= (1 << 21) # Apply sign extension by subtracting 2^21
248                     self.PC = self.PC + int(imm / 4) - 1

```

(continuing next page)

```

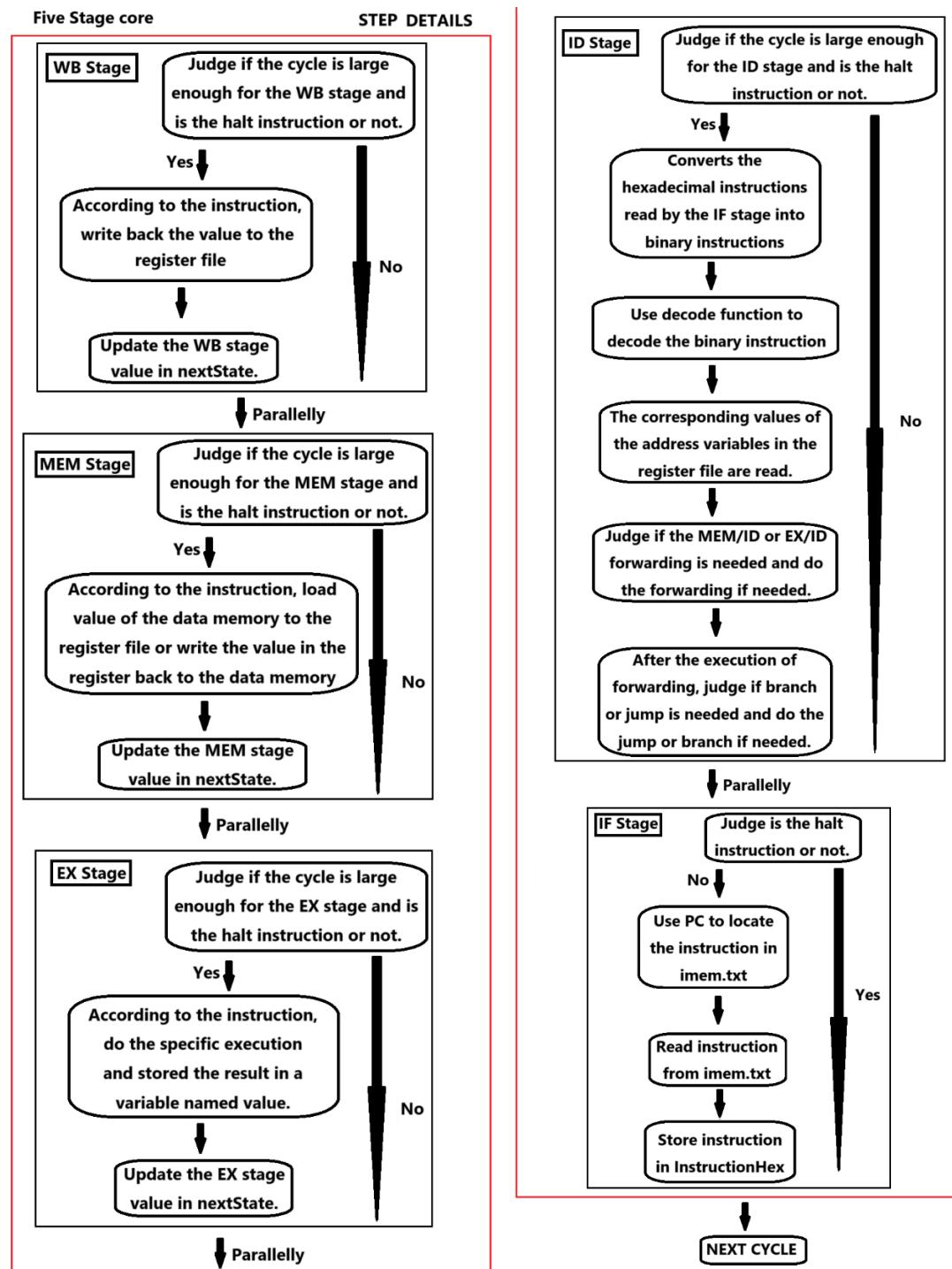
249     #MEM stage
250     if self.state.IF["nop"] == False:
251         if InstructionInfo['opcode'] in op_R:
252             if InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '000': # ADD
253                 value = int(self.myRF.readRF(rs1)) + int(self.myRF.readRF(rs2))
254             elif InstructionInfo['func7'] == '0100000' and InstructionInfo['func3'] == '000': # SUB
255                 value = int(self.myRF.readRF(rs1)) - int(self.myRF.readRF(rs2))
256             elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '100': # XOR
257                 value = int(self.myRF.readRF(rs1)) ^ int(self.myRF.readRF(rs2))
258             elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '110': # OR
259                 value = int(self.myRF.readRF(rs1)) | int(self.myRF.readRF(rs2))
260             elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '111': # AND
261                 value = int(self.myRF.readRF(rs1)) & int(self.myRF.readRF(rs2))
262
263         elif InstructionInfo['opcode'] in op_I:
264             if InstructionInfo['opcode'] == '0000011': # LW
265                 value = self.ext_dmem.readDataMem(rs1 + imm)
266             elif InstructionInfo['opcode'] == '0010011':
267                 if InstructionInfo['func3'] == '000': # ADDI
268                     value = int(self.myRF.readRF(rs1)) + imm
269                 elif InstructionInfo['func3'] == '100': # XORI
270                     value = int(self.myRF.readRF(rs1)) ^ imm
271                 elif InstructionInfo['func3'] == '110': # ORI
272                     value = int(self.myRF.readRF(rs1)) | imm
273                 elif InstructionInfo['func3'] == '111': # ANDI
274                     value = int(self.myRF.readRF(rs1)) & imm
275
276             elif InstructionInfo['opcode'] == op_S: # SW
277                 value = self.myRF.readRF(rs2)
278
279         elif InstructionInfo['opcode'] == op_B: #BEQ & BNE
280             pass
281
282         elif InstructionInfo['opcode'] == op_U:
283             pass
284
285         elif InstructionInfo['opcode'] == op_J: # JAL
286             value = (self.PC - int(imm / 4) + 1)*4 + 4
287
288     #WB stage
289     if self.state.IF["nop"] == False:
290         if InstructionInfo['opcode'] in op_R or InstructionInfo['opcode'] in op_I:
291             self.myRF.writeRF(rd, value)
292
293         elif InstructionInfo['opcode'] == op_S: # SW
294             self.ext_dmem.writeDataMem(address, value)
295
296         elif InstructionInfo['opcode'] == op_B: # BNE & BEQ
297             pass
298
299         elif InstructionInfo['opcode'] == op_U:
300             pass
301
302         elif InstructionInfo['opcode'] == op_J: # JAL
303             self.myRF.writeRF(rd, value)
304             self.instructionCount += 1
305
306     # -----
307     self.myRF.outputRF(self.cycle) # dump RF
308     if not self.state.IF["nop"]:
309         self.PC += 1
310     self.nextState.IF["PC"] = self.PC*4
311     print("\n")
312     self.printState(self.nextState, self.cycle) # print states after executing cycle 0, cycle 1, cycle 2 ...
313     self.state = self.nextState #The end of the cycle and updates the current state with the values calculated in this cycle
314     self.cycle += 1
315
316     if self.state.IF["nop"] and self.halted:
317         print("-----Single Stage Core Performance Metrics-----\n")
318         print("Number of cycles taken: " + str(self.cycle) + "\n")
319         print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
320         print("Cycles per instruction: " + str(self.cycle / self.instructionCount) + "\n")
321         print("Instructions per cycle: " + str(self.instructionCount / self.cycle) + "\n")
322
323     def printState(self, state, cycle): 1个用法
324         printstate = [".*70+" + "\n", "State after executing cycle: " + str(cycle) + "\n"]
325         printstate.append("TE PC: " + str(state.TE["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

```
6     class InsMem(object): 1个用法
7         def __init__(self, name, ioDir):
8             self.id = name
9             with open(ioDir + "\\imem.txt") as im:
10                 self.IMem = [data.replace(_old: "\n", _new: "") for data in im.readlines()]
11
12         def readInstr(self, ReadAddress): 2用法
13             #read instruction memory
14             #return 32 bit hex val
15             indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
16             selected_strings = [self.IMem[i] for i in indices_to_select]
17             for i in range(4):
18                 selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
19             instruction = ''.join(selected_strings)
20             return instruction
21
```

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

```
22     class DataMem(object): 2用法
23         def __init__(self, name, ioDir):
24             self.id = name
25             self.ioDir = ioDir
26             self.DMem = ["00000000"]*MemSize
27             with open(ioDir + "\\dmem.txt") as dm:
28                 self.DMem = [data.replace(_old: "\n", _new: "") for data in dm.readlines()]
29             self.DMem.extend(["00000000"] * (MemSize - len(self.DMem)))
30
31         def readDataMem(self, ReadAddress): 2个用法(2个动态)
32             #read data memory
33             #return 32 bit hex val
34             indices_to_select = [ReadAddress, ReadAddress + 1, ReadAddress + 2, ReadAddress + 3]
35             selected_strings = [self.DMem[i] for i in indices_to_select]
36             for i in range(4):
37                 selected_strings[i] = hex(int(selected_strings[i], 2))[2:].zfill(2)
38             DataMem = ''.join(selected_strings)
39             is_negative = DataMem[0] == '1'
40             if is_negative:
41                 inverted_binary = ''.join('1' if b == '0' else '0' for b in DataMem) # 取反
42                 positive_value = int(inverted_binary, 2) + 1 # 加1, 得到正数值
43                 DataMem = -positive_value # 加符号
44             else:
45                 DataMem = int(DataMem, 2)
46             print("data in mem is "+str(DataMem))
47             return DataMem
48
49         def writeDataMem(self, Address, WriteData): 2个用法(2个动态)
50             # write data into byte addressable memory
51             if WriteData < 0:
52                 binary_str = bin((1 << 32) + WriteData)[2:] # 加 2^32 实现补码
53             else:
54                 binary_str = bin(WriteData)[2:]
55             binaryData = binary_str.zfill(32)
56             binary_8bit_Data = [binaryData[i:i + 8] for i in range(0, 32, 8)]
57             required_lines = Address + 4
58             while len(self.DMem) < required_lines:
59                 self.DMem.append("\n")
60             for i in range(4):
61                 self.DMem[Address + i] = binary_8bit_Data[i]
62
63         def outputDataMem(self): 2用法
64             resPath = self.ioDir + "\\" + self.id + "_DMEMResult.txt"
65             print(self.DMem)
66             with open(resPath, "w") as rp:
67                 i = 0
68                 for data in self.DMem:
69                     if data != "\n" and i != len(self.DMem)-1:
70                         rp.writelines(str(data)+"\n")
71                     else:
72                         rp.writelines(str(data))
73                     i += 1
74
75     class RegisterFile(object): 1个用法
```

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

```
75     class RegisterFile(object): 1个用法
76         def __init__(self, ioDir):
77             self.outputFile = ioDir + "RFResult.txt"
78             binary_str = format(0, '032b')
79             self.Registers = [binary_str for i in range(32)]
80
81         def readRF(self, Reg_addr): 21用法
82             DataReg = self.Registers[Reg_addr]
83
84             is_negative = DataReg[0] == '1'
85             if is_negative:
86                 inverted_binary = ''.join('1' if b == '0' else '0' for b in DataReg) # 取反
87                 positive_value = int(inverted_binary, 2) + 1 # 加1, 得到正数值
88                 DataReg = -positive_value # 加符号
89             else:
90                 DataReg = int(DataReg, 2)
91
92             return DataReg
93
94         def writeRF(self, Reg_addr, Wrt_reg_data): 4用法
95             if Wrt_reg_data < 0:
96                 binary_str = bin((1 << 32) + Wrt_reg_data)[2:] # 加 2^32 实现补码
97             else:
98                 binary_str = bin(Wrt_reg_data)[2:]
99                 binaryData = binary_str.zfill(32)
100                self.Registers[Reg_addr] = binaryData
101
102        def outputRF(self, cycle): 2用法
103            op = ["-70+" + "\n", "State of RF after executing cycle:" + str(cycle) + "\n"]
104            op.extend([str(val) + "\n" for val in self.Registers])
105            if(cycle == 0): perm = "w"
106            else: perm = "a"
107            with open(self.outputFile, perm) as file:
108                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:

```
436         def WriteBack(InstructionInfo, RegAddr, value):
437             rd = RegAddr['rd']
438             if self.halted == False:
439                 if InstructionInfo['opcode'] in op_R or InstructionInfo['opcode'] in op_I:
440                     self.myRF.writeRF(rd, value)
441
442                 elif InstructionInfo['opcode'] == op_S: # SW
443                     pass
444                 elif InstructionInfo['opcode'] == op_B: # BNE & BEQ
445                     pass
446                 elif InstructionInfo['opcode'] == op_U:
447                     pass
448                 elif InstructionInfo['opcode'] == op_J: # JAL
449                     self.myRF.writeRF(rd, value)
```

Logic:

```
775         # -----WB Stage-----
776         if self.cycle > 4 and self.InstructionHexSet[str(self.WBIndex-1)] == 'ffffffff':
777             self.WBstop = True
778             self.state.WB["nop"] = True
779             self.nextState.WB["nop"] = True
780             print("WB STOP!")
781         elif self.cycle > 3 and not self.state.WB["nop"] and not self.WBstop:
782             WBIndex = str(self.WBIndex)
783             print("WB STAGE: " + WBIndex)
784             stop = False
785             if self.InstructionHexSet[str(self.WBIndex)] == 'ffffffff':
786                 stop = True
787             if not stop:
788                 WriteBack(self.InstructionSet[WBIndex], self.RegAddrSet[WBIndex], self.ValueSet[WBIndex])
789                 self.nextState.WB["Wrt_data"] = self.ValueSet[WBIndex]# Update WB state
790                 self.nextState.WB["Rs"] = self.RegAddrSet[WBIndex]['Rs']
791                 self.nextState.WB["Rt"] = self.RegAddrSet[WBIndex]['Rt']
792                 self.nextState.WB["Wrt_reg_addr"] = self.RegAddrSet[WBIndex]['rd']
793                 self.nextState.WB["wrt_enable"] = 1
794
795                 self.WBIndex = self.WBIndex+1
796             elif self.state.WB["nop"] and not self.WBstop:
797                 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:

```
800     # -----MEM Stage-----
801     if self.cycle > 3 and self.InstructionHexSet[str(self.MEMIndex-1)] == 'ffffffff':
802         self.MEMstop = True
803         self.state.MEM["nop"] = True
804         self.nextState.MEM["nop"] = True
805         print("MEM STOP!")
806
807     if self.cycle > 2 and not self.state.MEM["nop"] and not self.MEMstop:
808         MEMIndex = str(self.MEMIndex)
809         print("MEM STAGE: " + MEMIndex)
810         stop = False
811         if self.InstructionHexSet[str(self.MEMIndex)] == 'ffffffff':
812             stop = True
813         if not stop:#self.InstructionHexSet[str(self.IFIndex-1)] != 'ffffffff':
814             self.ValueSet[MEMIndex] = MemoryAccess(self.InstructionSet[MEMIndex], self.RegAddrSet[MEMIndex],
815                                                     self.ValueSet[MEMIndex])
816             updateMEMNextState()
817             print("after MEM, value is "+str(self.nextState.MEM["ALUresult"]))
818             self.MEMIndex = self.MEMIndex + 1
819
820         elif self.state.MEM["nop"] and not self.MEMstop:
821             self.nextState.MEM["nop"] = False
822             self.nextState.WB["nop"] = True
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
```

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:

```
525     def Execution(InstructionInfo, RegAddr):
526         value = None
527         if InstructionInfo is not None:
528             rs1 = RegAddr['rs1']
529             rs2 = RegAddr['rs2']
530             imm = RegAddr['imm']
531             if self.halted == False:
532                 if InstructionInfo['opcode'] in op_R:
533                     if InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '000': # ADD
534                         value = int(rs1) + int(rs2)
535                     elif InstructionInfo['func7'] == '0100000' and InstructionInfo['func3'] == '000': # SUB
536                         value = int(rs1) - int(rs2)
537                     elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '100': # XOR
538                         value = int(rs1) ^ int(rs2)
539                     elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '110': # OR
540                         value = int(rs1) | int(rs2)
541                     elif InstructionInfo['func7'] == '0000000' and InstructionInfo['func3'] == '111': # AND
542                         value = int(rs1) & int(rs2)
543
544                 elif InstructionInfo['opcode'] in op_I:
545                     if InstructionInfo['opcode'] == '0000011': # LW
546                         value = int(rs1) + int(imm)
547                     elif InstructionInfo['opcode'] == '0010011':
548                         if InstructionInfo['func3'] == '000': # ADDI
549                             value = int(rs1) + int(imm)
550                             print("R4 IS VALUE OF: "+str(value))
551                         elif InstructionInfo['func3'] == '100': # XORI
552                             value = int(rs1) ^ int(imm)
553                         elif InstructionInfo['func3'] == '110': # ORI
554                             value = int(rs1) | int(imm)
555                         elif InstructionInfo['func3'] == '111': # ANDI
556                             value = int(rs1) & int(imm)
557
558                 elif InstructionInfo['opcode'] == op_S: # SW
559                     value = int(rs2)
560
561                 elif InstructionInfo['opcode'] == op_B: # BEQ & BNE
562                     value = None
563
564                 elif InstructionInfo['opcode'] == op_U:
565                     pass
566
567                 elif InstructionInfo['opcode'] == op_J: # JAL
568                     value = (self.JALPC-1) * 4 + 4
569                     print("Value of JAL IN rd is "+str(value))
570
571             if InstructionInfo is None:
572                 value = None
573
574         return value
```

updateEXNextState:

```
574     def updateEXNextState():
575         if self.InstructionSet[str(self.EXIndex)] is not None:
576             if self.InstructionSet[str(self.EXIndex)][‘opcode’] in op_R:
577                 self.nextState.EX[“Read_data1”] = self.RegAddrSet[str(self.EXIndex)][‘rs1’]
578                 self.nextState.EX[“Read_data2”] = self.RegAddrSet[str(self.EXIndex)][‘rs2’]
579                 self.nextState.EX[“Imm”] = self.RegAddrSet[str(self.EXIndex)][‘imm’]
580                 self.nextState.EX[“Rs”] = self.RegAddrSet[str(self.EXIndex)][‘Rs’]
581                 self.nextState.EX[“Rt”] = self.RegAddrSet[str(self.EXIndex)][‘Rt’]
582                 self.nextState.EX[“Wrt_reg_addr”] = self.RegAddrSet[str(self.EXIndex)][‘rd’]
583                 self.nextState.EX[“is_I_type”] = False
584                 self.nextState.EX[“wrt_enable”] = 1
585                 #self.nextState.EX[“alu_op”] = “10”
586                 self.nextState.EX[“alu_op”] = self.ValueSet[str(self.EXIndex)]
587                 self.nextState.EX[“rd_mem”] = 0
588                 self.nextState.EX[“wrt_mem”] = 0
589
590             elif self.InstructionSet[str(self.EXIndex)][‘opcode’] in op_I:
591                 self.nextState.EX[“Read_data1”] = self.RegAddrSet[str(self.EXIndex)][‘rs1’]
592                 self.nextState.EX[“Read_data2”] = 0
593                 self.nextState.EX[“Imm”] = self.RegAddrSet[str(self.EXIndex)][‘imm’]
594                 self.nextState.EX[“Rs”] = self.RegAddrSet[str(self.EXIndex)][‘Rs’]
595                 self.nextState.EX[“Rt”] = 0
596                 self.nextState.EX[“Wrt_reg_addr”] = self.RegAddrSet[str(self.EXIndex)][‘rd’]
597                 self.nextState.EX[“is_I_type”] = True
598                 self.nextState.EX[“wrt_enable”] = 1
599                 # self.nextState.EX[“alu_op”] = “10”
600                 self.nextState.EX[“alu_op”] = self.ValueSet[str(self.EXIndex)]
601                 self.nextState.EX[“rd_mem”] = 0
602                 self.nextState.EX[“wrt_mem”] = 0
603                 if self.InstructionSet[str(self.EXIndex)][‘opcode’] == ‘00000011’: # LW
604                     self.nextState.EX[“rd_mem”] = 1 # need to read mem
605                     #self.nextState.EX[“alu_op”] = “00”
606
607             elif self.InstructionSet[str(self.EXIndex)][‘opcode’] in op_S:
608                 self.nextState.EX[“Read_data1”] = self.RegAddrSet[str(self.EXIndex)][‘rs1’]
609                 self.nextState.EX[“Read_data2”] = self.RegAddrSet[str(self.EXIndex)][‘rs2’]
610                 self.nextState.EX[“Imm”] = self.RegAddrSet[str(self.EXIndex)][‘imm’]
611                 self.nextState.EX[“Rs”] = self.RegAddrSet[str(self.EXIndex)][‘Rs’]
612                 self.nextState.EX[“Rt”] = self.RegAddrSet[str(self.EXIndex)][‘Rt’]
613                 self.nextState.EX[“Wrt_reg_addr”] = 0
614                 self.nextState.EX[“is_I_type”] = False
615                 # self.nextState.EX[“alu_op”] = “00”
616                 self.nextState.EX[“alu_op”] = self.ValueSet[str(self.EXIndex)]
617                 self.nextState.EX[“wrt_enable”] = 0
618                 self.nextState.EX[“rd_mem”] = 0
619                 self.nextState.EX[“wrt_mem”] = 1 # need to write mem
620
621             elif self.InstructionSet[str(self.EXIndex)][‘opcode’] in op_B:
622                 self.nextState.EX[“Read_data1”] = self.RegAddrSet[str(self.EXIndex)][‘rs1’]
623                 self.nextState.EX[“Read_data2”] = self.RegAddrSet[str(self.EXIndex)][‘rs2’]
624                 self.nextState.EX[“Imm”] = self.RegAddrSet[str(self.EXIndex)][‘imm’]
625                 self.nextState.EX[“Rs”] = self.RegAddrSet[str(self.EXIndex)][‘Rs’]
626                 self.nextState.EX[“Rt”] = self.RegAddrSet[str(self.EXIndex)][‘Rt’]
627                 self.nextState.EX[“Wrt_reg_addr”] = 0
628                 self.nextState.EX[“is_I_type”] = False
629                 # self.nextState.EX[“alu_op”] = “01”
630                 self.nextState.EX[“alu_op”] = self.ValueSet[str(self.EXIndex)]
631                 self.nextState.EX[“wrt_enable”] = 0
632                 self.nextState.EX[“rd_mem”] = 0
633                 self.nextState.EX[“wrt_mem”] = 0
634
635             elif self.InstructionSet[str(self.EXIndex)][‘opcode’] in op_J:
636                 self.nextState.EX[“Read_data1”] = 0
637                 self.nextState.EX[“Read_data2”] = 0
638                 self.nextState.EX[“Imm”] = self.RegAddrSet[str(self.EXIndex)][‘imm’]
639                 self.nextState.EX[“Rs”] = 0
640                 self.nextState.EX[“Rt”] = 0
641                 self.nextState.EX[“Wrt_reg_addr”] = self.RegAddrSet[str(self.EXIndex)][‘rd’]
642                 self.nextState.EX[“is_I_type”] = False
643                 # self.nextState.EX[“alu_op”] = “10”
644                 self.nextState.EX[“alu_op”] = self.ValueSet[str(self.EXIndex)]
645                 self.nextState.EX[“wrt_enable”] = 1
646                 self.nextState.EX[“rd_mem”] = 0
647                 self.nextState.EX[“wrt_mem”] = 0
```

Logic:

```

824     # -----EX Stage-----
825     if str(self.EXIndex-1) in self.InstructionSet and self.InstructionHexSet[str(self.EXIndex-1)] == 'ffffffff':
826         self.EXstop = True
827         self.state.EX["nop"] = True
828         self.nextState.EX["nop"] = True
829         print("EX STOP!")
830
831     if self.cycle > 1 and not self.state.EX["nop"] and not self.EXstop:
832         EXIndex = str(self.EXIndex)
833         print("EX STAGE: " + EXIndex)
834         stop = False
835         if self.InstructionHexSet[str(self.EXIndex)] == 'ffffffff':
836             stop = True
837         if not stop:
838             self.ValueSet[EXIndex] = Execution(self.InstructionSet[EXIndex], self.RegAddrSet[EXIndex])
839             updateEXNextState()
840             self.EXIndex = self.EXIndex + 1
841             self.nextState.MEM["nop"] = False
842         elif self.state.EX["nop"] and not self.EXstop:
843             self.nextState.EX["nop"] = False
844             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:

```

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

```

decode:

```

395     def decode(self, instruction): 一个用法
396         instructionInfo_R = {'func7': '', 'rs2': '', 'rs1': '', 'func3': '', 'rd': '', 'opcode': ''} # type
397         instructionInfo_I = {'imm[11:0]': '', 'rs1': '', 'func3': '', 'rd': '', 'opcode': ''}
398         instructionInfo_S = {'imm[11:5]': '', 'rs2': '', 'rs1': '', 'func3': '', 'imm[4:0]': '', 'opcode': ''}
399         instructionInfo_B = {'imm[12,10:5]': '', 'rs2': '', 'rs1': '', 'func3': '', 'imm[4:1,11]': '', 'opcode': ''}
400         instructionInfo_J = {'imm[20,10:1,11,19:12]': '', 'rd': '', 'opcode': ''}
401         instructionInfo = {}
402         opcode = instruction[-7:]
403         op_R = ('0110011', '0111011')
404         op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
405         op_S = '0100011'
406         op_B = '1100011'
407         op_U = '0110111'
408         op_J = '1101111'
409         if opcode in op_R:
410             instructionInfo_R['func7'] = instruction[:7]
411             instructionInfo_R['rs2'] = instruction[7:12]
412             instructionInfo_R['rs1'] = instruction[12:17]
413             instructionInfo_R['func3'] = instruction[17:20]
414             instructionInfo_R['rd'] = instruction[20:25]
415             instructionInfo_R['opcode'] = instruction[-7:]
416             instructionInfo = instructionInfo_R
417         elif opcode in op_I:
418             instructionInfo_I['imm[11:0]'] = instruction[:12]
419             instructionInfo_I['rs1'] = instruction[12:17]
420             instructionInfo_I['func3'] = instruction[17:20]
421             instructionInfo_I['rd'] = instruction[20:25]
422             instructionInfo_I['opcode'] = instruction[-7:]
423             instructionInfo = instructionInfo_I
424         elif opcode == op_S:
425             instructionInfo_S['imm[11:5]'] = instruction[:7]
426             instructionInfo_S['rs2'] = instruction[7:12]
427             instructionInfo_S['rs1'] = instruction[12:17]
428             instructionInfo_S['func3'] = instruction[17:20]
429             instructionInfo_S['imm[4:0]'] = instruction[20:25]
430             instructionInfo_S['opcode'] = instruction[-7:]
431             instructionInfo = instructionInfo_S
432         elif opcode == op_B:
433             instructionInfo_B['imm[12,10:5]'] = instruction[:7]
434             instructionInfo_B['rs2'] = instruction[7:12]
435             instructionInfo_B['rs1'] = instruction[12:17]
436             instructionInfo_B['func3'] = instruction[17:20]
437             instructionInfo_B['imm[4:1,11]'] = instruction[20:25]
438             instructionInfo_B['opcode'] = instruction[-7:]
439             instructionInfo = instructionInfo_B
440         elif opcode == op_J:
441             instructionInfo_J['imm[20,10:1,11,19:12]'] = instruction[:20]
442             instructionInfo_J['rd'] = instruction[20:25]
443             instructionInfo_J['opcode'] = instruction[-7:]
444             instructionInfo = instructionInfo_J
445         elif opcode == '1111111':
446             instructionInfo = None
447         else:
448             print(f"something wrong with the instruction decoding!!!")
449         return instructionInfo

```

RegRead:

```

479     def RegRead(InstructionInfo):
480         op_R = ('0110011', '0111011')
481         op_I = ('0010011', '0000011', '0001111', '0011011', '1100111', '1110011')
482         op_S = '0100011'
483         op_B = '1100011'
484         op_J = '1101111'
485         rs1 = 0
486         rs2 = 0
487         rd = 0
488         imm = 0
489
490         if InstructionInfo is None:
491             Regread = None
492         elif InstructionInfo is not None:
493             if InstructionInfo['opcode'] in op_R:
494                 rs1 = int(InstructionInfo['rs1'], 2)
495                 rs2 = int(InstructionInfo['rs2'], 2)
496                 rd = int(InstructionInfo['rd'], 2)
497
498             elif InstructionInfo['opcode'] in op_I:
499                 rs1 = int(InstructionInfo['rs1'], 2)
500                 rd = int(InstructionInfo['rd'], 2)
501                 imm_str = InstructionInfo['imm[11:0]']
502                 if imm_str[0] == '1':
503                     imm = int(imm_str, 2) - (1 < len(imm_str))
504                 else:
505                     imm = int(imm_str, 2)
506
507             elif InstructionInfo['opcode'] == op_S:
508                 rs1 = int(InstructionInfo['rs1'], 2)
509                 rs2 = int(InstructionInfo['rs2'], 2)
510                 imm_str = InstructionInfo['imm[4:0]']
511                 imm = int(imm_str, 2)
512
513             elif InstructionInfo['opcode'] == op_B: # BEQ & BNE
514                 rs1 = int(InstructionInfo['rs1'], 2)
515                 rs2 = int(InstructionInfo['rs2'], 2)
516                 imm_binary = (InstructionInfo['imm[12,10:5]'][0] + InstructionInfo['imm[4:1,11]'][4] +
517                               InstructionInfo['imm[12,10:5]'][1:7] + InstructionInfo['imm[4:1,11]'][4:8] + '0')
518                 imm = int(imm_binary, 2)
519                 # Sign extension for 13-bit immediate (if the 33th bit is 1, the value is negative)
520                 if imm_binary[6] == '1': # Check if the highest bit (sign bit) is 1
521                     imm -= (1 << 13) # Apply sign extension by subtracting 2^13
522
523             elif InstructionInfo['opcode'] == op_J: # JAL
524                 rd = int(InstructionInfo['rd'], 2)
525                 imm_binary = (InstructionInfo['imm[20,10:1,11,19:12]'][0] + InstructionInfo['imm[20,10:1,11,19:12]'] +
526                               InstructionInfo['imm[20,10:1,11,19:12]'][11] + InstructionInfo['imm[20,10:1,11,19:12]'][11:20])
527                 imm = int(imm_binary, 2)
528                 # Sign extension for 21-bit immediate (if the 21th bit is 1, the value is negative)
529                 if imm_binary[0] == '1': # Check if the highest bit (sign bit) is 1
530                     imm -= (1 << 21) # Apply sign extension by subtracting 2^21
531                 self.JALPC = self.PC
532                 self.PC = self.PC + int(imm / 4) - 2
533
534                 self.state.IF['nop'] = True
535                 self.nextState.ID['nop'] = True
536                 print("stop IF here for JAL and next State ID stop!!!!!!!!!!!!!!")
537
538             Regread = {'rs1': self.myRF.readRF(rs1), 'rs2': self.myRF.readRF(rs2), 'rd': rd, 'imm': imm,
539                       'Ra': rs1, 'Rt': rs2}
540             # judge if forwarding for ID is needed and forwarding realization below-----
541             def NOTBType(X):
542                 if X in (op_B, op_S):
543                     return False
544                 else:
545                     return True
546
547             if self.InstructionSet[str(self.IDIndex)][['opcode']] in (tuple(op_R) + (op_B, op_S)) and (not self.LUHSy):
548                 if self.IDIndex<8 and NOTBType(self.InstructionSet[str(self.IDIndex-1)][['opcode']]) and rs1 == self:
549                     Regread['rs1'] = self.nextState.EX['alu_op']
550                     print("forwarding for rs1 from EX with value: "+str(Regread['rs1']))
551                 if self.IDIndex>8 and NOTBType(self.InstructionSet[str(self.IDIndex-2)][['opcode']]) and rs1 == self:
552                     Regread['rs1'] = self.nextState.MEM["ALUresult"]
553                     print("forwarding for rs1 from MEM with value: "+str(Regread['rs1']))
554                 if self.IDIndex<8 and NOTBType(self.InstructionSet[str(self.IDIndex-1)][['opcode']]) and rs2 == self:
555                     Regread['rs2'] = self.nextState.EX['alu_op']
556                     print("forwarding for rs2 from EX with value: "+str(Regread['rs2']))
557                 if self.IDIndex>8 and NOTBType(self.InstructionSet[str(self.IDIndex-2)][['opcode']]) and rs2 == self:
558                     Regread['rs2'] = self.nextState.MEM["ALUresult"]
559                     print("forwarding for rs2 from MEM with value: "+str(Regread['rs2']))
560
561             elif self.InstructionSet[str(self.IDIndex)][['opcode']] in op_I and (not self.LUHSy):
562                 if self.IDIndex<8 and NOTBType(self.InstructionSet[str(self.IDIndex-1)][['opcode']]) and rs1 == self:
563                     Regread['rs1'] = self.nextState.EX['alu_op']
564                     print("forwarding for rs1 from EX")
565                 if self.IDIndex>8 and NOTBType(self.InstructionSet[str(self.IDIndex-2)][['opcode']]) and rs1 == self:
566                     Regread['rs1'] = self.nextState.MEM["ALUresult"]
567                     print("forwarding for rs1 from MEM")
568
569             elif self.InstructionSet[str(self.IDIndex)][['opcode']] in op_J:
570                 pass
571
572             elif self.InstructionSet[str(self.IDIndex)][['opcode']] in (tuple(op_R) + (op_B, op_S)) and self.LUHSy:
573                 if self.IDIndex<8 and (self.InstructionSet[str(self.IDIndex-1)][['opcode']]=='0000011') and rs1 == self:
574                     Regread['rs1'] = self.nextState.MEM["ALUresult"]
575                     print("forwarding for rs1 from MEM with value: "+str(Regread['rs1']))
576                 if self.IDIndex>8 and (self.InstructionSet[str(self.IDIndex-1)][['opcode']]=='0000011') and rs2 == self:
577                     Regread['rs2'] = self.nextState.MEM["ALUresult"]
578                     print("forwarding for rs2 from MEM with value: "+str(Regread['rs2']))
579
580             if InstructionInfo['opcode'] == op_B: # BEQ & BNE branch after forwarding
581                 if InstructionInfo['func3'] == '000':
582                     if Regread['rs1'] == Regread['rs2']:
583                         self.PC = self.PC + int(imm / 4) - 2
584                         self.state.IF['nop'] = True
585                         self.nextState.ID['nop'] = True
586                         print("stop IF here for BEQ and next State ID stop!!!!!!!!!!!!!!")
587                     elif Regread['rs1'] != Regread['rs2']:
588                         self.PC = self.PC + int(imm / 4) - 2
589                         self.state.IF['nop'] = True
590                         self.nextState.ID['nop'] = True
591                         print("stop IF here for BNE and next State ID stop!!!!!!!!!!!!!!")
592
593             return Regread
594
595     >     def InstructionFetch():...

```

Logic:

```
846     # -----ID Stage-----
847     if self.cycle > 1 and self.InstructionHexSet[str(self.IDIndex-1)] == 'ffffffff':
848         self.IDstop = True
849         self.state.ID["nop"] = True
850         self.nextState.ID["nop"] = True
851         self.nextState.ID["Instr"] = self.state.ID["Instr"]
852         print("ID STOP!")
853     elif self.cycle > 0 and not self.state.ID["nop"] and not self.IDstop:
854         IDIndex = str(self.IDIndex)
855         if not self.state.ID["nop"]:
856             print("ID STAGE: " + IDIndex)
857             InstructionH = self.InstructionHexSet[IDIndex]
858             self.InstructionSet[IDIndex] = InstructionDecode(InstructionH)
859             if (self.IDIndex>0 and (self.InstructionSet[str(self.IDIndex-1)]['opcode']=='00000011') and
860                 ('rd' in self.InstructionSet[str(self.IDIndex-1)]) and
861                 ('rs1' in self.InstructionSet[IDIndex] and self.InstructionSet[IDIndex]['rs1'] == self.Instruc-
862                 ('rs2' in self.InstructionSet[IDIndex] and self.InstructionSet[IDIndex]['rs2'] == self.Instruc-
863                 (not self.LUHSyet)):# Load-Use Hazard Detection
864                 print("Load Use Hazard!!!!!!!!!!!!!!")
865                 self.nextState.ID["nop"] = False
866                 self.nextState.EX["nop"] = True
867                 self.LUHSyet = True
868             else:
869                 self.RegAddrSet[IDIndex] = RegRead(self.InstructionSet[IDIndex])
870                 self.LUHSyet = False
871                 print(self.RegAddrSet[IDIndex])
872                 if self.InstructionSet[str(self.IDIndex)] is None:
873                     self.nextState.ID["Instr"] = self.state.ID["Instr"]
874                 elif self.InstructionSet[str(self.IDIndex)] is not None:
875                     self.nextState.ID["Instr"] = bin(int(self.InstructionHexSet[str(self.IDIndex)], 16))[2:].zfill(32)
876                     self.IDIndex = self.IDIndex + 1
877
878
879             elif self.state.ID["nop"]:
880                 self.nextState.ID["nop"] = False
881                 self.nextState.EX["nop"] = True
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
```

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:

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

Logic:

```
883     #-----IF Stage-----
884     if self.cycle > 0 and self.InstructionHexSet[str(self.IFIndex-1)] == 'ffffffff':
885         self.IFstop = True
886         self.state.IF["nop"] = True
887         self.nextState.IF["nop"] = True
888         print("IF STOP!")
889     elif not self.state.IF["nop"] and not self.IFstop:
890         IFIndex = str(self.IFIndex)
891         InstructionHex = InstructionFetch()
892         self.InstructionHexSet[IFIndex] = InstructionHex
893         print("IF STAGE: " + str(self.IFIndex) + " " + InstructionHex)
894         self.IFIndex = self.IFIndex + 1
895     if self.state.IF["nop"]:
896         self.nextState.IF["nop"] = False
897         self.nextState.IF["PC"] = (self.PC+1)*4
898
```

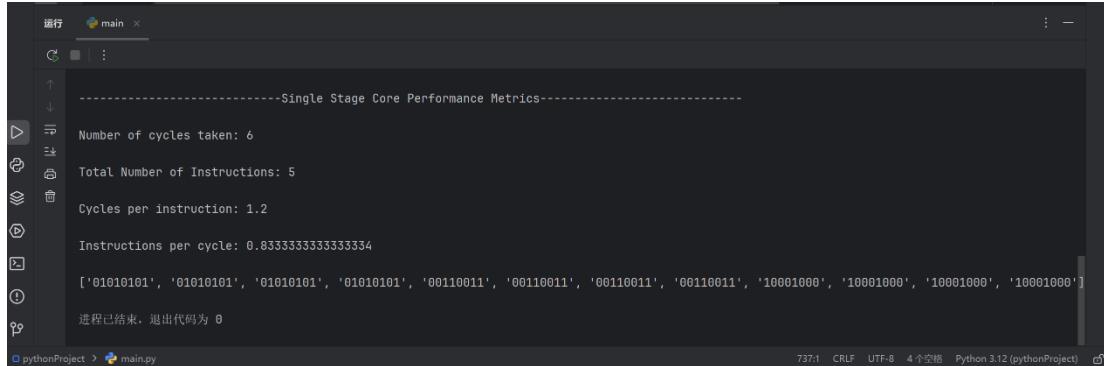
The class FiveStageCore(Core) is implemented as follows:

Task 3:

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

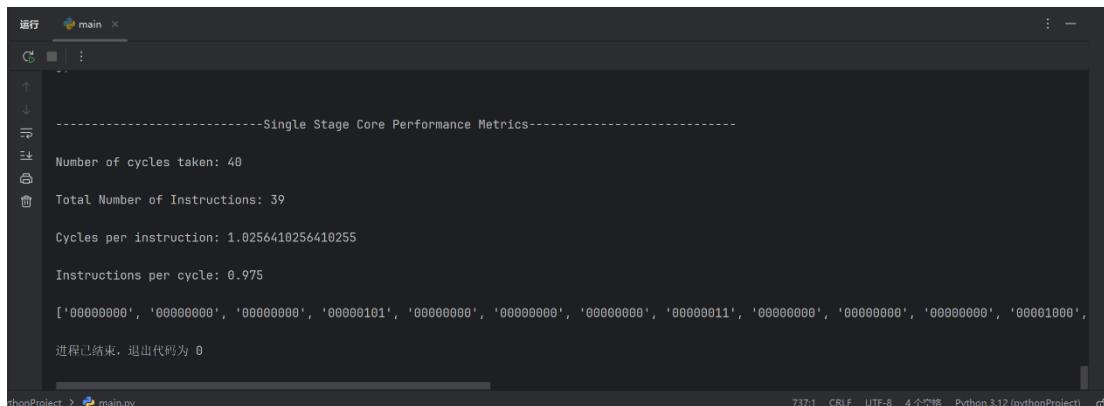
```
315     if self.state.IF["nop"] and self.halted:
316         print("-----Single Stage Core Performance Metrics-----\n")
317         print("Number of cycles taken: " + str(self.cycle) + "\n")
318         print("Total Number of Instructions: " + str(self.instructionCount) + "\n")
319         print("Cycles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



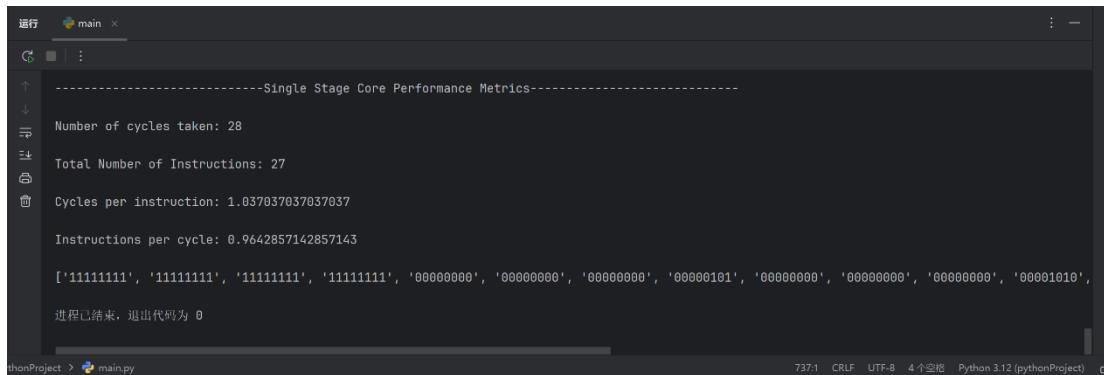
```
-----Single Stage Core Performance Metrics-----
Number of cycles taken: 6
Total Number of Instructions: 5
Cycles per instruction: 1.2
Instructions per cycle: 0.8333333333333334
['01010101', '01010101', '01010101', '01010101', '00110011', '00110011', '00110011', '10001000', '10001000', '10001000']
进程已结束, 退出代码为 0
```

The output matrix for testcase1 is:



```
-----Single Stage Core Performance Metrics-----
Number of cycles taken: 40
Total Number of Instructions: 39
Cycles per instruction: 1.0256410256410255
Instructions per cycle: 0.975
['00000000', '00000000', '00000000', '00000101', '00000000', '00000000', '00000011', '00000000', '00000000', '00000000', '00001000',
进程已结束, 退出代码为 0
```

The output matrix for testcase2 is:



```
-----Single Stage Core Performance Metrics-----
Number of cycles taken: 28
Total Number of Instructions: 27
Cycles per instruction: 1.037037037037037
Instructions per cycle: 0.9642857142857143
['11111111', '11111111', '11111111', '11111111', '00000000', '00000000', '00000101', '00000000', '00000000', '00000000', '00001010',
进程已结束, 退出代码为 0
```

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:

$$ET = \text{number of cycles} \times \text{cycle time}$$

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**.