ECE 2162 Part1 Report

Shixin Ji & Xingzhen Chen

Section 1

1. The code is written in python, and we test them on Windows via VSCode;

2. The source code is in “src” folder, and the main file is named by “main.py”, and input document is named by “input.txt”, which is placed in “input” folder.

3. To execute the code, git clone https://github.com/XingzhenCHEN/ECE2162_CopmterArc.git, and run the main function in the main file.

4. The output of the program will be displayed on the terminal screen, and includes instruction cycle count, ARF, memory value, and it will also display the detail logging of each cycle.

Section 2

Initializing as below:

	# of RS	Cycles in EX	Cycles in Mem	# of FUs
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

Instruction queue size = 10

# of integer register(R<i>) = 8

# of float register(F<i>) = 8

ROB entry = 64

CDB buffer entry = 1

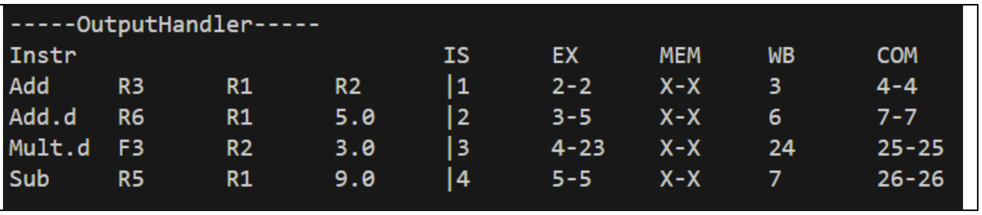
R1 = 10, R2 = 20, F2 = 30.1

Mem[4] = 1, Mem[8] = 2, Mem[12] = 3.4

For coherent of this report, all test cases shown in this report have the same initial configuration and values.

Case 1: straight-line, fundamental instruction, no dependency

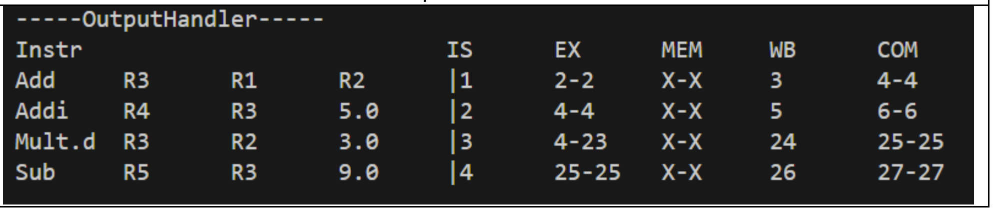
Instructions	Output Screenshot

Add R3,R1,R2 Add.d R6,R1,5 Mult.d F3,R2,3 Sub R5,R1,9	

```
-----OutputHandler-----
Instr                                   IS        EX        MEM       WB        COM
Add       R3        R1        R2        |1        2-2       X-X       3         4-4
Add.d     R6        R1        5.0       |2        3-5       X-X       6         7-7
Mult.d    F3        R2        3.0       |3        4-23      X-X       24        25-25
Sub       R5        R1        9.0       |4        5-5       X-X       7         26-26
```

It is the simplest instructions, and it can test our modules and main functions. According to the output information, the body part of our project is complete.

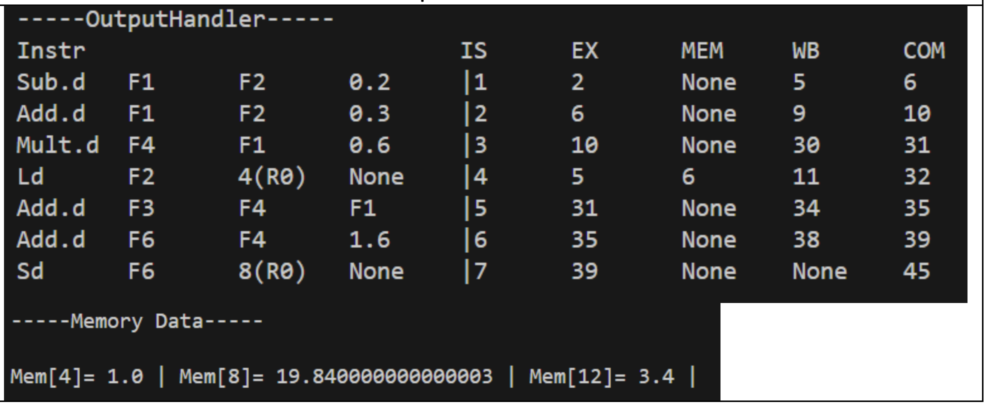
Case 2: straight-line, fundamental instruction, true dependency

Instructions	Output Screenshot
Add R3,R1,R2 Addi R4,R3,5 Mult.d R3,R2,3 Sub R5,R3,9	

```
-----OutputHandler-----
Instr                                   IS        EX        MEM       WB        COM
Add       R3        R1        R2        |1        2-2       X-X       3         4-4
Addi      R4        R3        5.0       |2        4-4       X-X       5         6-6
Mult.d    R3        R2        3.0       |3        4-23      X-X       24        25-25
Sub       R5        R3        9.0       |4        25-25     X-X       26        27-27
```

This test consists of both false and true dependencies among Integer registers. As we can see, in the four instructions, there are always true dependencies in R3 register, so Addi instruction has to wait until Add instruction writeback and get R3 value, so the EX stage of Addi has to wait until cycle 4 to execute, though it is issued in cycle 2. The same as before, the Sub instruction has to wait until cycle 25 to execute, though it is issued in cycle 4.

Case 3: straight-line, complex instruction, Load/Store, true dependency

Instructions	Output Screenshot
Sub.d F1,F2,0.2 Add.d F1,F2,0.3 Mult.d F4,F1,0.6 Ld F2,4(R0) Add.d F3,F4,F1 Add.d F6,F4,1.6 Sd F6,8(R0)	

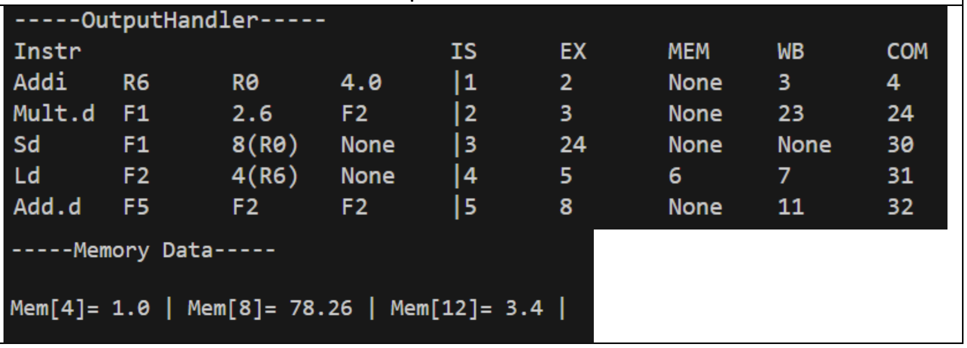
```
-----OutputHandler-----
Instr                                   IS        EX        MEM       WB        COM
Sub.d     F1        F2        0.2       |1        2         None      5         6
Add.d     F1        F2        0.3       |2        6         None      9         10
Mult.d    F4        F1        0.6       |3        10        None      30        31
Ld        F2        4(R0)     None      |4        5         6         11        32
Add.d     F3        F4        F1        |5        31        None      34        35
Add.d     F6        F4        1.6       |6        35        None      38        39
Sd        F6        8(R0)     None      |7        39        None      None      45

-----Memory Data-----

Mem[4]= 1.0 | Mem[8]= 19.840000000000003 | Mem[12]= 3.4 |
```

In this testcase, we add the load and store function to the instructions, and we focus on the Ld and Sd instructions. In Ld instruction, when the PU calculates the memory address, it will go to the memory to load the data, which will cost 5 cycles, and after 5 cycles, it will begin to writeback and wait for commit. In the Sd instruction, there is a true dependency in F6 register, so we has to wait until the last instruction finished and writeback the F6 value to go to the store stage in commit stage.

Case 4: straight-line, forwarding among load/store instructions

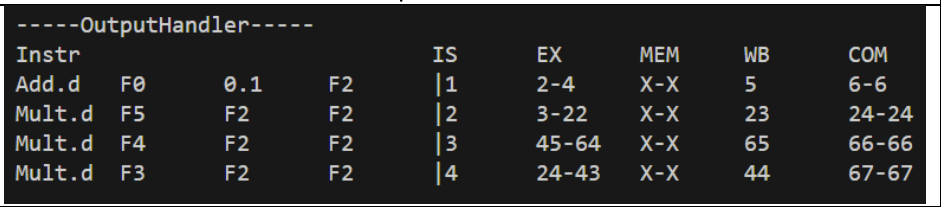
Instructions	Output Screenshot
Addi R6,R0,4 Mult.d F1,2.6,F2 Sd F1,8(R0) Ld F2,4(R6) Add.d F5,F2,F2	

```
-----OutputHandler-----
Instr                                 IS       EX       MEM      WB       COM
Addi     R6       R0       4.0       |1        2        None     3        4
Mult.d   F1       2.6      F2        |2        3        None     23       24
Sd       F1       8(R0)    None      |3        24       None     None     30
Ld       F2       4(R6)    None      |4        5        6        7        31
Add.d    F5       F2       F2        |5        8        None     11       32
-----Memory Data-----

Mem[4]= 1.0 | Mem[8]= 78.26 | Mem[12]= 3.4 |
```

In this testcase, the Ld instruction should go the memory[8] to fetch the data, but in the previous load and store queue, there is a Sd instruction to store F1 value to the same memory address, which is also memory[8], so this instruction will not go to memory again to fetch the data from memory, which will cost 5 cycles, and it just forward the memory[8] value from the previous Ld/Sd queue to F2 register.

Case 5: straight-line, structural hazards in reservation stations and functional units

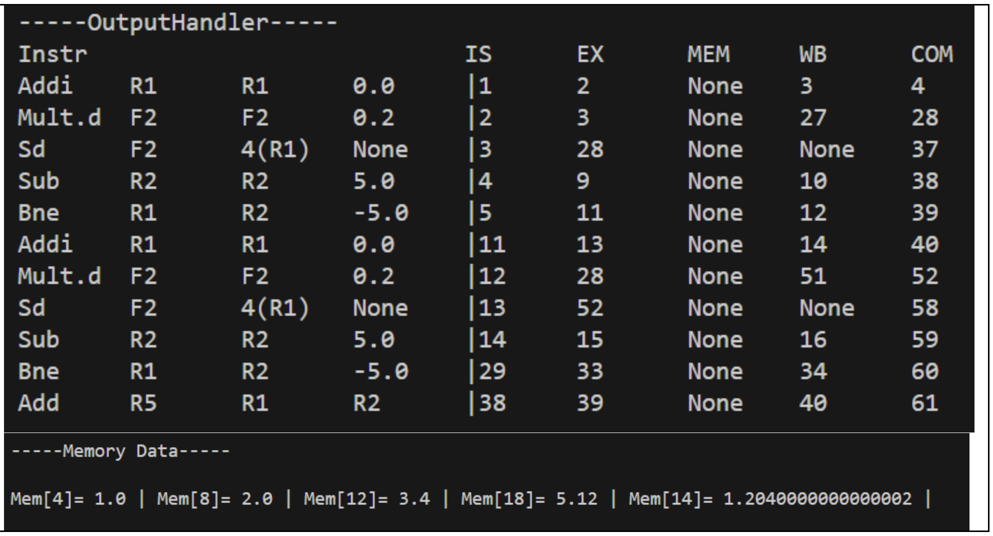
Instructions	Output Screenshot
Add.d F0,0.1,F2 Mult.d F5,F2,F2 Mult.d F4,F2,F2 Mult.d F3,F2,F2	

```
-----OutputHandler-----
Instr                                 IS       EX       MEM      WB       COM
Add.d    F0       0.1      F2        |1        2-4      X-X      5        6-6
Mult.d   F5       F2       F2        |2        3-22     X-X      23       24-24
Mult.d   F4       F2       F2        |3        45-64    X-X      65       66-66
Mult.d   F3       F2       F2        |4        24-43    X-X      44       67-67
```

In this testcase, since we only have 2 multipliers, the third instruction has to wait until the first instruction finished, and because of our strategy, the second Mult.d instruction should wait and in cycle 45, which means the third Mult.d instruction finished, it will enter the PU to execute, which is a hazard there.

Case 6: simple loop

Instructions	Output Screenshot

Addi R1,R1,0 Mult.d F2,F2,0.2 Sd F2,4(R1) Sub R2,R2,5 Bne R1,R2,-5 Add R5,R1,R2	

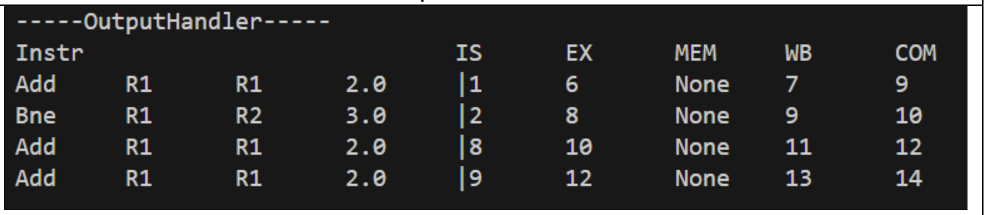
```
-----OutputHandler-----
Instr                             IS      EX      MEM     WB      COM
Addi    R1      R1      0.0      |1       2       None    3       4
Mult.d  F2      F2      0.2      |2       3       None    27      28
Sd      F2      4(R1)   None     |3       28      None    None    37
Sub     R2      R2      5.0      |4       9       None    10      38
Bne     R1      R2      -5.0     |5       11      None    12      39
Addi    R1      R1      0.0      |11      13      None    14      40
Mult.d  F2      F2      0.2      |12      28      None    51      52
Sd      F2      4(R1)   None     |13      52      None    None    58
Sub     R2      R2      5.0      |14      15      None    16      59
Bne     R1      R2      -5.0     |29      33      None    34      60
Add     R5      R1      R2       |38      39      None    40      61
```

```
-----Memory Data-----

Mem[4]= 1.0 | Mem[8]= 2.0 | Mem[12]= 3.4 | Mem[18]= 5.12 | Mem[14]= 1.2040000000000002 |
```

In this testcase, we add the Bne instruction to test the loop function, and according to the instructions, the Bne instruction will take once and not take once, so in the second loop, the PC will quit the loop and point to the next instruction, which is Add and finish the instructions.

Case 7: demonstration of branch prediction

Instructions	Output Screenshot
Add R1,R1,2 Bne R1,R2,3 Add R1,R1,2 Add R1,R1,2 Add R1,R1,2 Add R1,R1,2 Add R1,R1,2	

```
-----OutputHandler-----
Instr                             IS      EX      MEM     WB      COM
Add     R1      R1      2.0      |1       6       None    7       9
Bne     R1      R2      3.0      |2       8       None    9       10
Add     R1      R1      2.0      |8       10      None    11      12
Add     R1      R1      2.0      |9       12      None    13      14
```

In this testcase, we add the branch prediction to demonstrate. According to the instructions, the Bne instruction should be taken, and PC jump to the 6th instruction, but in our branch prediction, we predict it will be not taken, so we continue fetch the next instruction as below. But after the first instruction finishd, and the result comes out, it should be taken, so we roll back the system, and flush the ROB, PU, RS and so on, then we continue to execute the right instructions, so we can get the correct final output in the screenshot sheet.

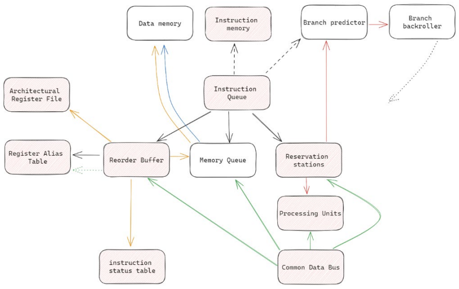
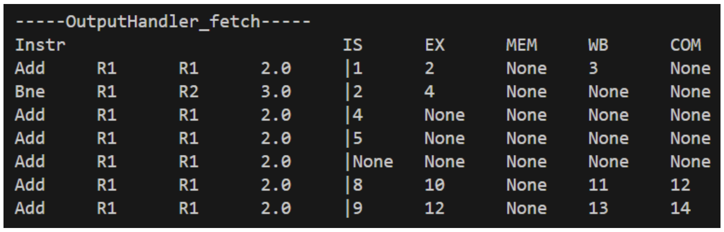


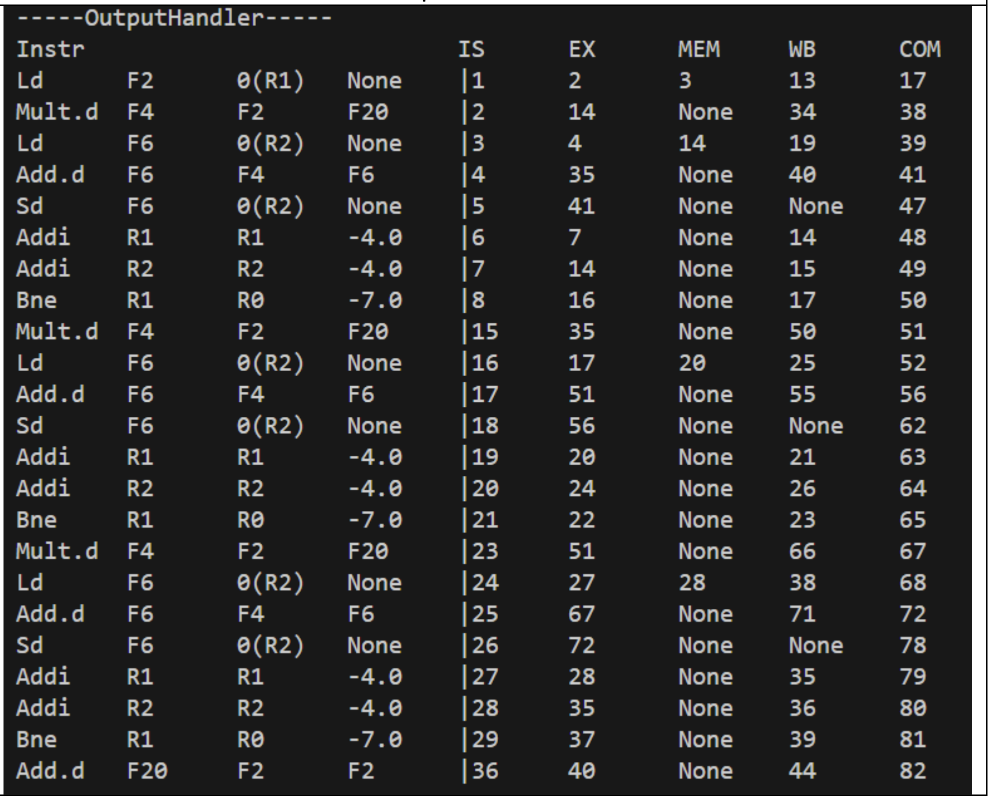
Figure 1 overall structure



```
-----OutputHandler_fetch-----
Instr                       IS      EX      MEM     WB      COM
Add     R1      R1      2.0      |1       2       None    3       None
Bne     R1      R2      3.0      |2       4       None    None    None
Add     R1      R1      2.0      |4       None    None    None    None
Add     R1      R1      2.0      |5       None    None    None    None
Add     R1      R1      2.0      |None    None    None    None    None
Add     R1      R1      2.0      |8       10      None    11      12
Add     R1      R1      2.0      |9       12      None    13      14
```

Figure 2: fetch instructions

Case 8: demo presentation

Instructions	Output Screenshot
Ld F2,0(R1) Mult.d F4,F2,F20 Ld F6,0(R2) Add.d F6,F4,F6 Sd F6,0(R2) Addi R1,R1,-4 Addi R2,R2,-4 Bne R1,R0,-7 Add.d F20,F2,F2	

```
-----OutputHandler-----
Instr                            IS      EX      MEM     WB      COM
Ld      F2      0(R1)   None    |1       2       3       13      17
Mult.d  F4      F2      F20     |2       14      None    34      38
Ld      F6      0(R2)   None    |3       4       14      19      39
Add.d   F6      F4      F6      |4       35      None    40      41
Sd      F6      0(R2)   None    |5       41      None    None    47
Addi    R1      R1      -4.0    |6       7       None    14      48
Addi    R2      R2      -4.0    |7       14      None    15      49
Bne     R1      R0      -7.0    |8       16      None    17      50
Mult.d  F4      F2      F20     |15      35      None    50      51
Ld      F6      0(R2)   None    |16      17      20      25      52
Add.d   F6      F4      F6      |17      51      None    55      56
Sd      F6      0(R2)   None    |18      56      None    None    62
Addi    R1      R1      -4.0    |19      20      None    21      63
Addi    R2      R2      -4.0    |20      24      None    26      64
Bne     R1      R0      -7.0    |21      22      None    23      65
Mult.d  F4      F2      F20     |23      51      None    66      67
Ld      F6      0(R2)   None    |24      27      28      38      68
Add.d   F6      F4      F6      |25      67      None    71      72
Sd      F6      0(R2)   None    |26      72      None    None    78
Addi    R1      R1      -4.0    |27      28      None    35      79
Addi    R2      R2      -4.0    |28      35      None    36      80
Bne     R1      R0      -7.0    |29      37      None    39      81
Add.d   F20     F2      F2      |36      40      None    44      82
```

In this demo case, since we use different roll back strategy, the timing would be worse than expectation. It is because when we find a false prediction, we would roll back all the instructions before commit and execute again, which will cost more cycles, but the correctness can be guaranteed.

Section 3

Group members: Xingzhen Chen, Shixin Ji.

Xingzhen Chen mainly focuses on the ARF, RAT, ROB, Memory, which is related to load and store queue, stage_writeback, stage_commit and main function modules, and comes up with these testcase, output handlers, debugging solutions and writing of the report.

Shixin Ji mainly focuses on the overall instruction design, Instr, Instr_queuem Instr_memory, PUs, RS, stage_init, stage_fetch, stage_issue, stage_execute, main functions and comes up with initial formats, branch prediction strategy and debugging solutions.