ASSIGNMENT-1 **(COA – README FILE)**

NAME: Methuku Preetham ROLL: 16CS01045

QUESTION 1

1) **Types of Dependencies:**
   a) **RAW** (Read After Write)
   b) **WAW** (Write After Write)
   c) **WAR** (Write After Write)

2) Initially, I am checking all the types of dependencies between instruction **i** and instruction **j** such that **j > i (for all j).**

3) I constructed a dependency acyclic graph with nodes as instruction number. (With 0 as starting number).

4) I constructed that dependency graph in order to **reorder the given instructions.**

5) If the dependency is **RAW,** then there is a need for inserting **NOPs.**

   For Example:

   Ins (0): ADD R1 R2 R3
   Ins (1): LOAD R3, #100(R1)

   - Here by seeing the above two instructions, we can say that there is a **RAW.**
   - The new value of **R1** comes at the end of **5th** stage.
   - Hence in order to prevent reading the old value, there is a need to insert **NOPs**.

   Initially→

   ```
   F D C M R
     F D C M R
   ```

   Finally→

   ```
   F D C M R (ADD)
     F D - - - (NOP)
       F D - - - (NOP)
         F D - - - (NOP)
           F D C M R (LOAD)
   ```

6) If the dependency is **WAW,** then there is no need for inserting **NOPs.**

   For Example:

   Ins (0): ADD R1 R2 R3
   Ins (1): LOAD R1, #100(R5)

   - Here by seeing the above two instructions, we can say that there is a **WAW.**
   - The new value of **R1** comes at the end of **5th** stage.
   - At that time, instruction(1) will be performing memory stage.

7) If the dependency is **WAR,** then there is no need for inserting **NOPs.**

For Example:

Ins (0): ADD R1 R2 R3
Ins (1): LOAD R2, #100(R5)

- Here by seeing the above two instructions, we can say that there is a **WAR.**
- The value of **R2** will be read during **2th** stage.
- At that time, instruction(1) will be performing fetch stage and writing the new value of **R2** will takes place at **5th** stage.

Conclusion:

- There is a need to insert **NOPs** whenever there is **RAW** dependency.
- I have printed all the dependencies as required for the **1st** question.

QUESTION 2

1) **Algorithm followed for reordering instructions :**

List all the vertices with indegree zero(0).
Repeat until the list is empty
{

- Initialize **prev** to -1;
- **(If prev != -1)** Add all the newly formed vertices with indegree zero(0) to the list.
- Shuffle the list using inbuilt **new Random()** function in java(which will shuffle according to **gaussian** distribution).
- The above step is implemented as it is observed by me that it is giving better result (with less **NOPs**)
- Now iterate over the list and emit that node (instruction) which does not have any connection with the **prev** (if it is not equal to -1) node and break from that loop as soon as you get one such node.
- If you don't get any such node, just emit that node which is minimum in the list.
- Now remove all the edges that are connected to the previously emitted node.
- Assign the value of freshly emitted node to **prev**.

}

2) This algorithm will make sure that the register values (at the end of all the instructions) will be same as before reordering the instructions.

3) There are many topological sorting orders possible for the constructed dependency graph.

4) The above algorithm will give one near to optimal **(less NOPs)** topological order.

5) The best optimal order is when one checks all the topological sorting orders and gives the best one.

6) **Unfortunately,** finding all the topological orders is a **NP-Complete problem.**

7) **Hence, I came up with a near to optimal solution with polynomial time O(N^2) solution.**

8) Maybe compiler also gives near to optimal solution in reality.

9) After reordering the given instructions, I maintained a list by inserting **NOPs** whenever necessary.

10) I checked the list dynamically.

For Example:

Ins (0): ADD R1 R2 R3
Ins (1): LOAD R2, #100(R1)
Ins (2): ADD R6 R7 R1

Here -
There is a **RAW dependency** between **ins 0** and **ins 1**
There is no dependency between **ins 1** and **ins 2**
There is a **RAW dependency** between **ins 0** and **ins 2**

If I don't check dynamically, it would be as follows:

```
ADD R1 R2 R3
NOP
NOP
NOP
LOAD R2, #100(R1)
NOP
NOP
ADD R6 R7 R1
```

I checked the list dynamically, so it is as follows:

```
ADD R1 R2 R3
NOP
NOP
NOP
LOAD R2, #100(R1)
ADD R6 R7 R1
```

QUESTION 3

1) Coming to memory delays, I am iterating through the final list.
2) If it is a **LOAD** or **STORE** instruction, I am checking whether the next instruction is **HLT** or **NOP** or other instruction.
3) If it is **HLT** or **NOP,** then clock cycles are not wasted due to memory delay of **LOAD** or **STORE** instruction.
4) If **NOP** or **HLT** instruction is present at 2^{nd} or 3^{rd} place after **LOAD** or **STORE** instruction, then 2 clock cycles are wasted.

For Example:

Ins (0): LOAD R1 #100(R3)
Ins (1): NOP
Ins (2): ADD R6 R7 R4
Ins (3): OR R9 R10 R11

```
F D C M R (LOAD)
  F  D  -  -  - (NOP)
     F  D C M R (ADD)
        F  D C M R (OR)
```

- Here in the memory stage, only 2 stages (Compute stage of **ADD** instruction and Fetch stage of **OR**) are present. Hence the count is **4.**

For Example:

Ins (0): LOAD R1 #100(R3)
Ins (1): ADD R6 R7 R4
Ins (2): NOP
Ins (3): OR R9 R10 R11

```
F D C M R (LOAD)
  F  D C M R (ADD)
     F  D  -  -  - (NOP)
        F  D C M R (OR)
```

- Here in the memory stage, 3 stages (Compute stage of **ADD** instruction, Decode stage of **NOP** and Fetch stage of **OR**) are present. Hence the count is **6**

CONCLUSION:

1) As far as I understood, getting the best optimal order without any change in meaning of code is **NP-Complete. (as there is a need to find all topological sorting orders)**
2) Hence, I came up with a near to optimal solution (**with less NOPs**) compared to brute force method in polynomial time **O(N^2).**