PART A

P3) Find the Greatest Common Divisor (GCD) of two chosen numbers

Logic Used

The Euclidean Algorithm to calculate the GCD was used. For this, a method to calculate the reminder of a division calculation was needed. The subroutine *rem* was used for this case. Repetitive reminder calculations were done using the subroutine *change* by setting the values of registers s0 and s1 to the new pair of numbers whose reminder was to be calculated.

Assembly Code 1

Optimised for Flush Instruction method of Branch Hazard Handling

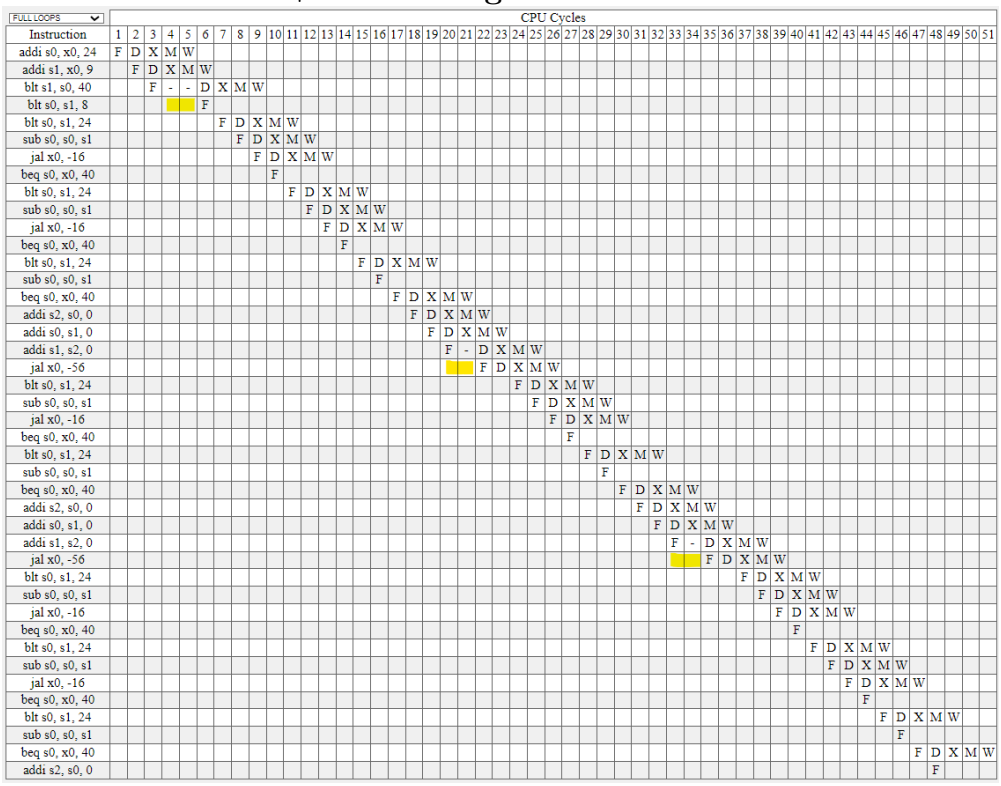
```
# inputs
addi s0, x0, 24
addi s1, x0, 9

check:
        #check for the lesser number and swap if needed
        blt s1, s0, rem
        blt s0, s1, swap

swap:
        addi s2, s0, 0
        addi s0, s1, 0
        addi s1, s2, 0

rem:
        blt s0, s1, change
        sub s0, s0, s1
        j rem

change:
        beq s0, x0, finish
        addi s2, s0, 0
        addi s0, s1, 0
        addi s1, s2, 0
        j rem

finish:
```

Assembly Code 2

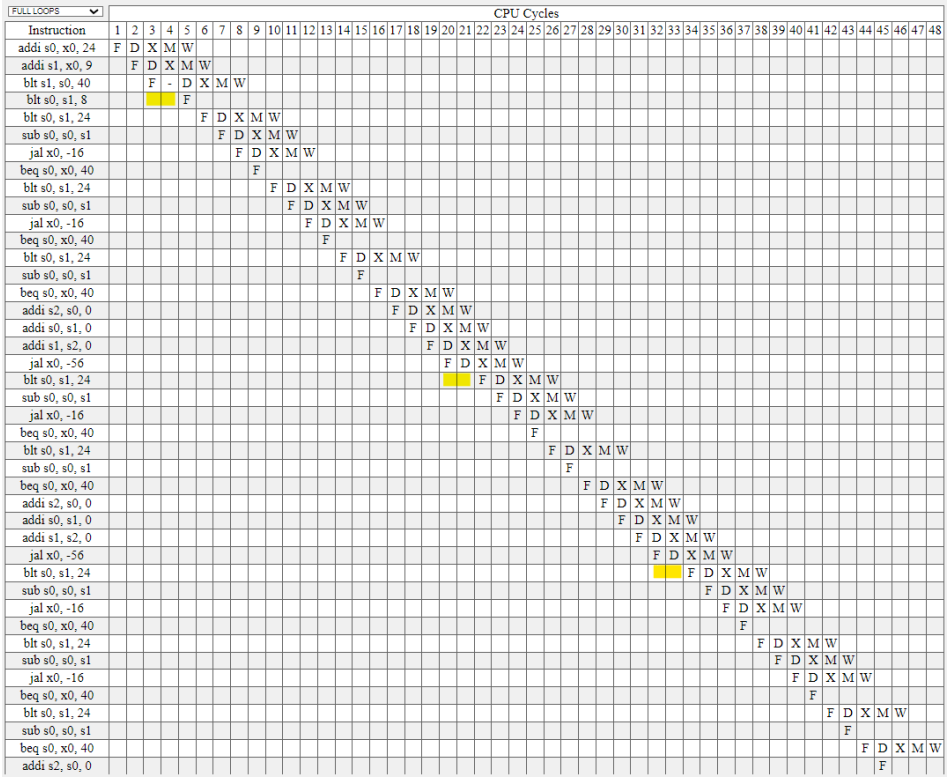
Optimised for Execute Delay Slot method of Branch Hazard Handling

```
# inputs
addi s0, x0, 24
addi s1, x0, 9

check:
        #check for the lesser number and swap if needed
        blt s1, s0, rem
        nop
        blt s0, s1, swap
        nop

swap:
        addi s2, s0, 0
        addi s0, s1, 0
        addi s1, s2, 0

rem:
        blt s0, s1, change
        nop
        j rem
        sub s0, s0, s1

change:
        beq s0, x0, finish
        nop
        addi s2, s0, 0
        addi s0, s1, 0
        j rem
        addi s1, s2, 0

finish:
```

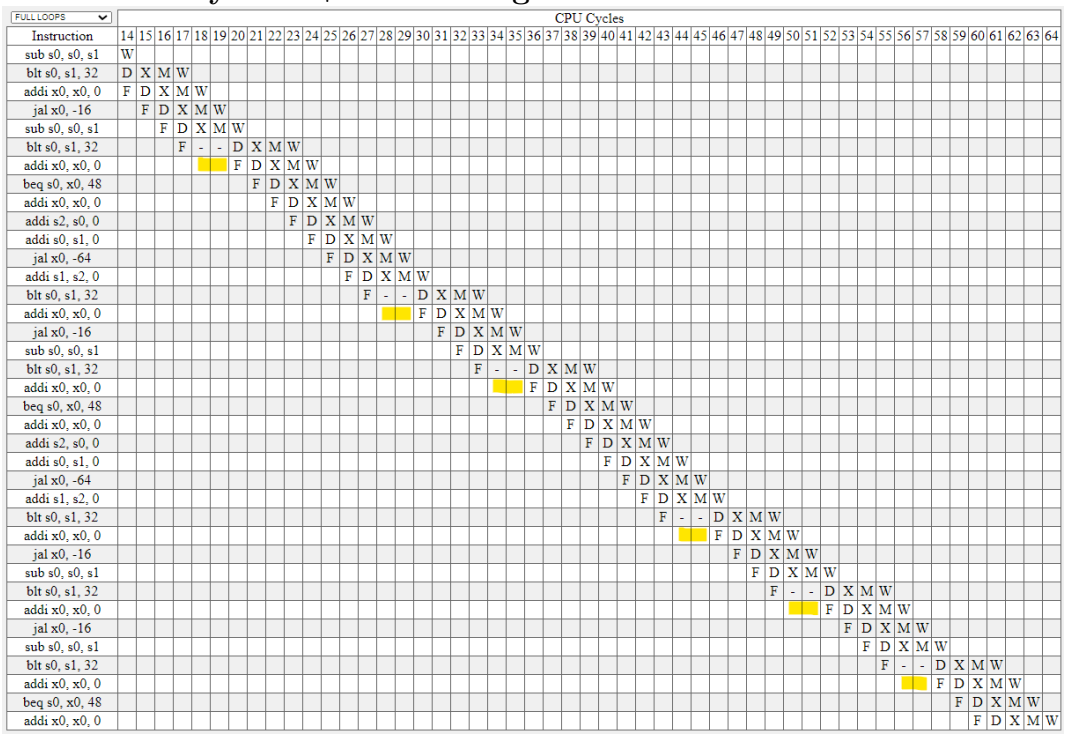
Flush Instruction + Forwarding Deactivated



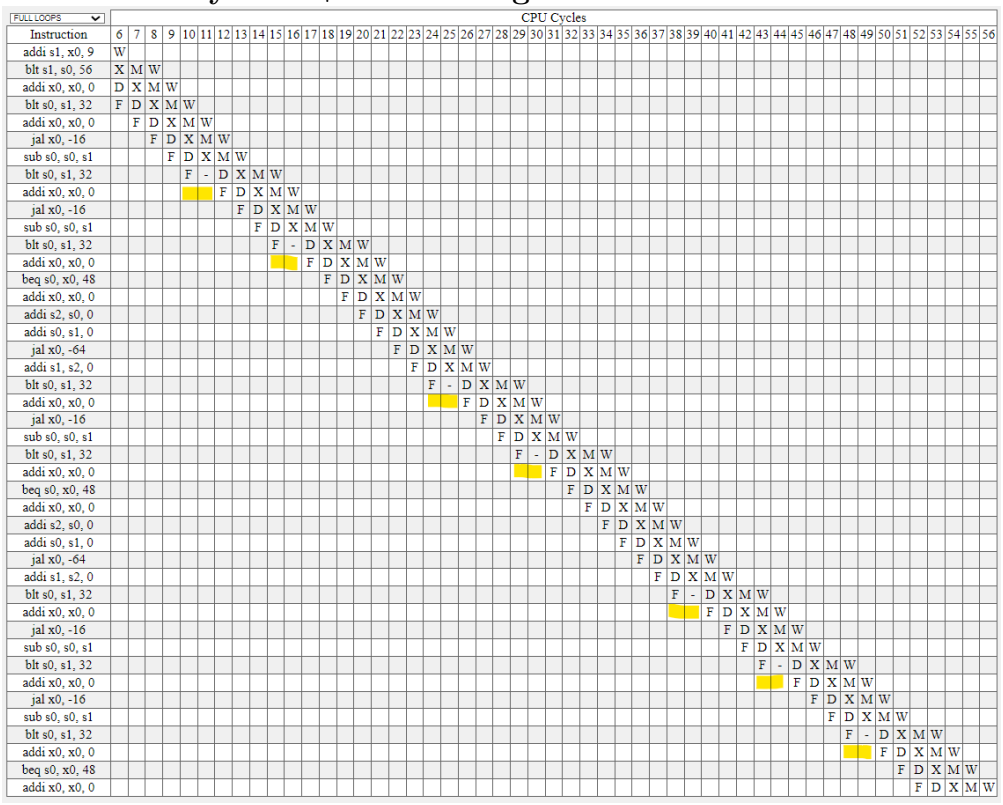
Flush Instruction + Forwarding Activated



Execute Delay Slot + Forwarding Deactivated



Execute Delay Slot + Forwarding Deactivated



Throughput Calculations

- Flush Instruction + Forwarding Deactivated = 42/51 = 0.823 instructions per cycle
- Flush Instruction + Forwarding Activated = 42/48 = 0.875 instructions per cycle
- Execute Delay Slot + Forwarding Deactivated = 37/64 = 0.578 instructions per cycle
- Execute Delay Slot + Forwarding Activated = 43/56 = 0.767 instructions per cycle

Discussions

When *flush instruction* is used as the branch hazard handling mechanism, we can see that stalls occur during data hazards. Without forwarding, it is clear that the stall lasts 2 clock cycles when an immediate data dependency is present since the next instruction must wait until the result is stored in memory before accessing it. Whereas, when forwarding is activated, only 1 clock cycle is stalled since the result is available after the execute stage. These results are clearly visible in the reservation tables shown above.
When *execute delay slot* is used as the branch handling mechanism, the code is slightly modified to reduce the number of clock cycles by replacing delay slots with instructions that are independent of the branch instruction itself. When this is not possible, *nop* instructions are placed in the branch delay slot. Forwarding activated and deactivated result in same stalls as the previous case.

PART B - Test Arithmetic

```
infernape@MSI:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/test_arithmetic.riscv -b AT
30
-10
370350
411
49380
771
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 519
Number of Cycles: 799
Avg Cycles per Instrcution: 1.5395
Branch Perdiction Accuacy: 0.5747 (Strategy: Always Taken)
Number of Control Hazards: 80
Number of Data Hazards: 235
Number of Memory Hazards: 13
-----------------------------------
```

```
infernape@MSI:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/test_arithmetic.riscv -b NT
30
-10
370350
411
49380
771
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 507
Number of Cycles: 751
Avg Cycles per Instrcution: 1.4813
Branch Perdiction Accuacy: 0.4268 (Strategy: Always Not Taken)
Number of Control Hazards: 91
Number of Data Hazards: 224
Number of Memory Hazards: 13
-----------------------------------
```

```
infernape@MSI:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/test_arithmetic.riscv -b BTFNT
30
-10
370350
411
49380
771
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 519
Number of Cycles: 749
Avg Cycles per Instrcution: 1.4432
Branch Perdiction Accuacy: 0.7701 (Strategy: Back Taken Forward Not Taken)
Number of Control Hazards: 63
Number of Data Hazards: 237
Number of Memory Hazards: 13
-----------------------------------
```

```
infernape@MSI:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/test_arithmetic.riscv -b BPB
30
-10
370350
411
49380
771
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 519
Number of Cycles: 784
Avg Cycles per Instrcution: 1.5106
Branch Perdiction Accuacy: 0.6322 (Strategy: Branch Prediction Buffer)
Number of Control Hazards: 75
Number of Data Hazards: 235
Number of Memory Hazards: 13
-----------------------------------
```

Tabulation of Results

Branch Strat.	CPI	Pred. Accuracy	Control Haz.	Data Haz.	Memory Haz.
AT	1.5395	0.5747	80	235	13
NT	1.4813	0.4268	91	224	13
BTFNT	1.4432	0.7701	63	237	13
BPB	1.5106	0.6322	75	235	13

Discussions

It is very evident that when BTFNT is used as the branch prediction strategy, the prediction accuracy is the highest. This could potentially mean that most of the branch statements were at the end of *for* loops. But, another observation is that when AT or NT is used as the branch prediction strategy, there is an accuracy of 0.574 and 0.426 respectively, which are significantly high and could imply that there are a lot of *for* loops which run for few iterations each, and the branch is present at the end of these *for* loops.
Another simple observation is that among the 4 prediction strategies, BTFNT results in the least number of Cycles Per Instruction (CPI), which directly translates from the fact that it has a higher prediction accuracy.