# Report

2021S50618,2021S10556

April 2023

# 1 PART-A

## 1.1 design decision

1. WB is done in the first half cycle, hence stalling is only done till a data-hazard value is in stages before WB.
2. we have implemented each stage of the 5-stage pipeline as separate struct, with pointers to its adjacent latches.
3. we have created latch between each stage such that in the next clock cycle the required value can be obtained from the latch.
4. whenever we have data hazard between the two instructions , we used stall to get the correct value
5. implemented an option for output format, using a variable outputFormat declared at the start of the MIPS_processor, when it is 0 the code will print what each stage is doing at each clock cycle and what stages are stalling, while when it is 1 the output will be of the format given by the TAs(used for autograding).
6. Control Signals and working: IF stage
   (a) if we have fetched all the given commands then no more instruction fetching will be done.our Control signal isWorking will be 0 that means no more instruction fetching.
   (b) if the Decode stage is stalling for some case that means we cannot fetch a new command so in this case we have to give our control signal isIDstalling as 1 which stalls the IF stage.
   (c) Beside the above two cases for the IF stage control signal isWorking is 1 always.
7. Control Signals and working: ID stage

(a) if we have fetched all the commands then the control signal isWorking for the ID stage will be 0 as we don't have any new commands to decode or if fetch is stalled then also the control signal isStalling in the decode will be 1 which stops the decode stage.

(b) we are checking for the data hazard in this stage and if there is any data hazard then we stall and give the control signal for the IF stage isIDstalling as 1 which stalls the Instruction fetching.

(c) we will send the datavalues from the read registers and the type of instruction and the write register into the latch between ID and EX as they are needed for the next stage.

8. Control signals and working : EX stage

(a) isWorking control signal of the decode stage is 0 that means we have completely fetched all the instruction and the isworking control signal for the Ex stage will also be 0.

(b) if the instruction in the latch between ID and EX is empty string then we don't have any operation to perform.

(c) if instruction is not empty then we will identity the instruction and will store the result of the operation and the write register in the latch between EX and DM.

(d) we are sending a control signal memwrite into the latch between EX and DM which will tell us either we have to read from the memory or write in memory.

(e) when the execution process for the beq and bne is completed then the new command can be fetched.

(f) if the instruction type is lw/sw, and the address calculated is not divisible by 4, then we output with cerr that address is not word aligned.

9. Control signals and working : DM stage

(a) from the latch between EX and DM we will read the control signal memwrite if the signal is 1 then we will write result calculated from the EX stage into the memory if memwrite control signal is 0 then load the data from the memory and passed this data and write register into the latch between DM and WB in the case of load.

(b) if the instruction is neither lw nor sw then we just don't do anything in this stage and ww simply put the data value and write register in the latch between DM and WB.

(c) if we are doing sw instruction or the write register is empty string then we pass write register as empty string in the latch between DM and WB.

10. Control signals and working : WB stage

(a) when the register string r1 is passed as "" or null, then WB stage does nothing. Otherwise it writes into the r1 register with the value that is passed to it from DM.

### 1.2 LOGIC FOR CHECKING OF DATA HAZARD AND STALLING

1. we are creating a map between the register(string) and its latch number(int).
2. in this every time in the decode stage if instruction is not I-type and branch type then we insert a pair of write register with latch number 2.
3. we update this map at each cycle as the values of the registers move to the next latch.
4. now in the next clock cycle we will check if the read register is in the data-hazard map and the latch number is less than 5, that means we have still not written the correct value in the read register, so we will stall the next ID and IF stage.

## 2 PART-B

### 2.1 design decision

1. bypassing is implemented using the same datahazard map as in part A, but with additional operations.
2. we are passing a vector from ID to the L3 latch to EX to L4 latch to DM, and we fill this vector in ID based on which latch the data-hazard register is in. this vector becomes a new control signal for EX and DM
3. the EX and DM stage then read this vector for each input register and determine from which latch they should take their values from, L3, L4 or L5.
4. this implementation is in accordance with hardware systems, and can be implemented using multiplexers (each value of the vector that is passed essentially controls the multiplexer for each read register).

### 2.2 LOGIC FOR FORWARDING

1. we proceed in the same way as we have done in part-1 dependency checking but in this case we have modified our map , it is now a map of register(string) with a pair of (latch number,instruction-type) both are integers.

2. the map lets us know which latch the required value is in currently, and we use this in the ID stage and pass a vector to L3 latch containing the latch numbers of where the value needs to be taken from.

3. then EX and DM stages can read this vector and appropriately decide if they should take values from other latches.

### 2.3 Observation between part-1 and part-2

When data forwarding is used, the results of an instruction that has not yet been stored in memory are forwarded directly to the input of the next instruction that needs it, instead of waiting for the result to be written back to memory and then read again. This reduces the number of clock cycles required to execute the dependent instruction, since it does not have to wait for the result to be written back and then read from memory.

As a result, forwarding can reduce the number of clock cycles required to execute instructions, which in turn can improve the overall performance of the CPU. By minimizing the number of clock cycles required to execute instructions, forwarding can help to reduce the latency of the CPU and increase its throughput, allowing it to perform more work in a given amount of time.

## 3 PART-C

### 3.1 Design Decisions

1. 2 instructions can be in the WB stage simultaneously if only one of them needs to use the write port. hence instructions like 'sw' and 'add' can be in the same WB stage, while 'lw' and 'add' cannot.

2. a 7stage command that follows a 9stage command is stalled until we can ensure that either they will leave the writeback stage together as in the above point or 9stage instruction will leave it first if they both use the write port.

3. similar structs are made for each stage and latch, as for 5stage.

#### 3.1.1 Logic for no-bypass

The way stalling is implemented in this one is different from 5stage. here we are declaring two global integers that are used to control stalls, branchStall and stallNumber. if branchStall or stallNumber = k, then the first k stages are stalled. BranchStall is used whenever we encounter a beq/bne/j instruction, we increment branchStall and pass the values to the next stage so in the next cycle, the current stage will be stalling until BranchStall is reset to 0.

Similarly, when ID1 detects a dataHazard, it simply updates the value of stallNumber to 3 so the first 3 stages are stalled, and then returns and checks the dataHazard condition in the next cycle.

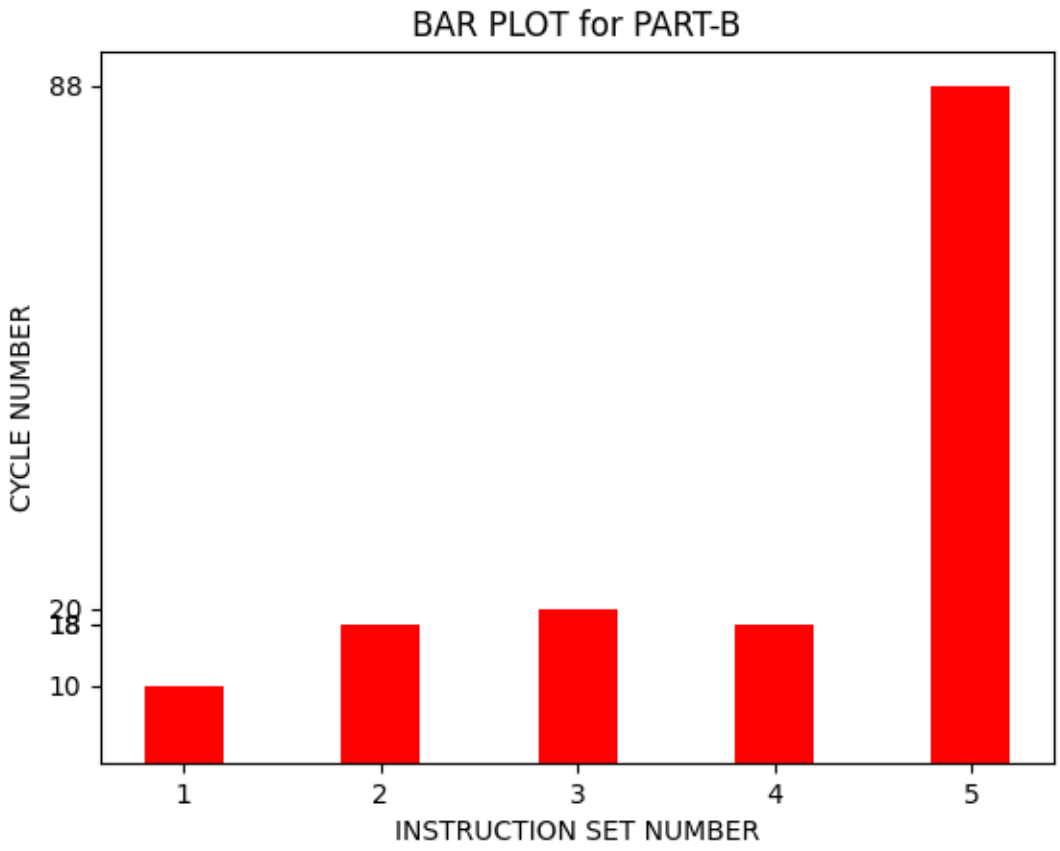
| | |
|---|---|
| test-1 | 21 |
| test-2 | 36 |
| test-3 | 32 |
| test-4 | 34 |
| test-5 | 150 |

### 3.2 BAR PLOT FOR PART-3

## 4 PART-E

### 4.1 Branch prediction Accuracies table

| file name with initial state | Prediction strategy | Accuracies |
|---|---|---|
| cnt00.txt | 2-bit Saturating Counter | 0.790146 |
| cnt01.txt | 2-bit Saturating Counter | 0.839416 |
| cnt10.txt | 2-bit Saturating Counter | 0.879562 |
| cnt11.txt | 2-bit Saturating Counter | 0.866788 |
| bhr00.txt | branch History register | 0.715328 |
| bhr01.txt | branch History register | 0.722628 |
| bhr10.txt | branch History register | 0.726277 |
| bhr11.txt | branch History register | 0.728102 |
| satbhr00.txt | saturatingBHR | 0.813869 |
| satbhr01.txt | saturatingBHR | 0.855839 |
| satbhr10.txt | saturatingBHR | 0.870438 |
| satbhr11.txt | saturatingBHR | 0.854015 |


BAR PLOT for PART-B
88
20
18
10
CYCLE NUMBER
1
2
3
4
5
INSTRUCTION SET NUMBER

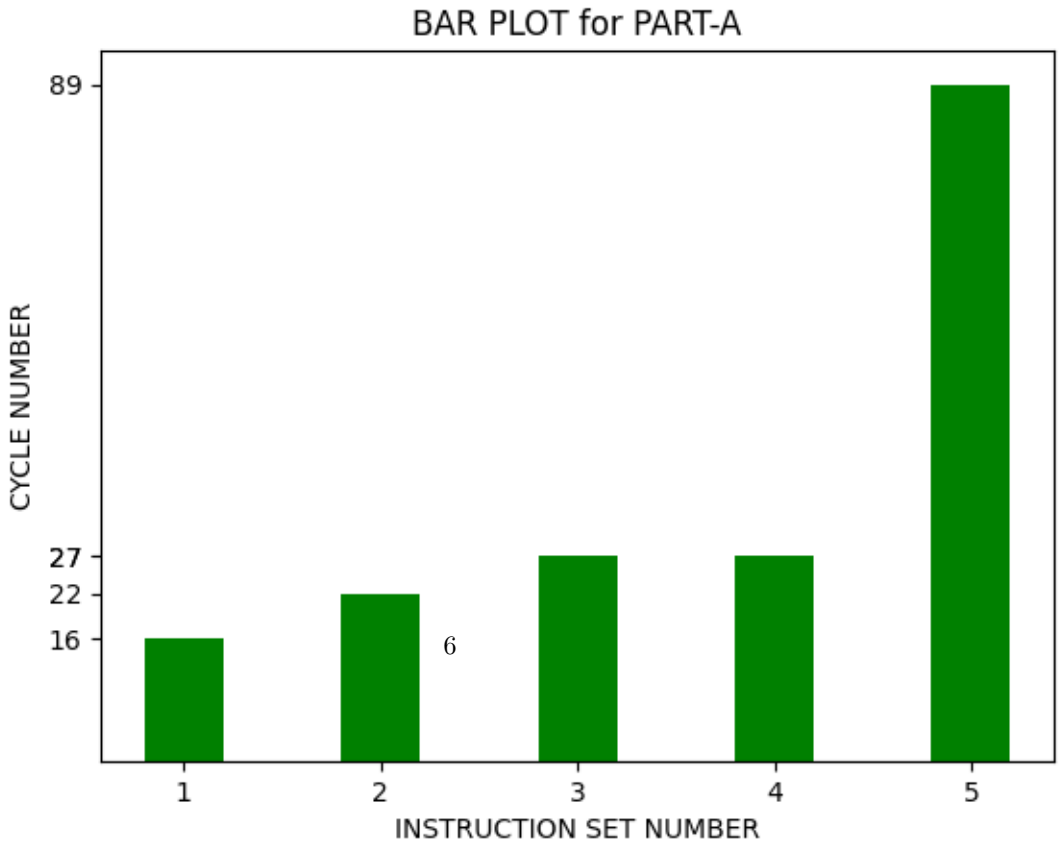

BAR PLOT for PART-A
89
27
22
16
CYCLE NUMBER
1
2
3
4
5
INSTRUCTION SET NUMBER

Figure 2: PART-1

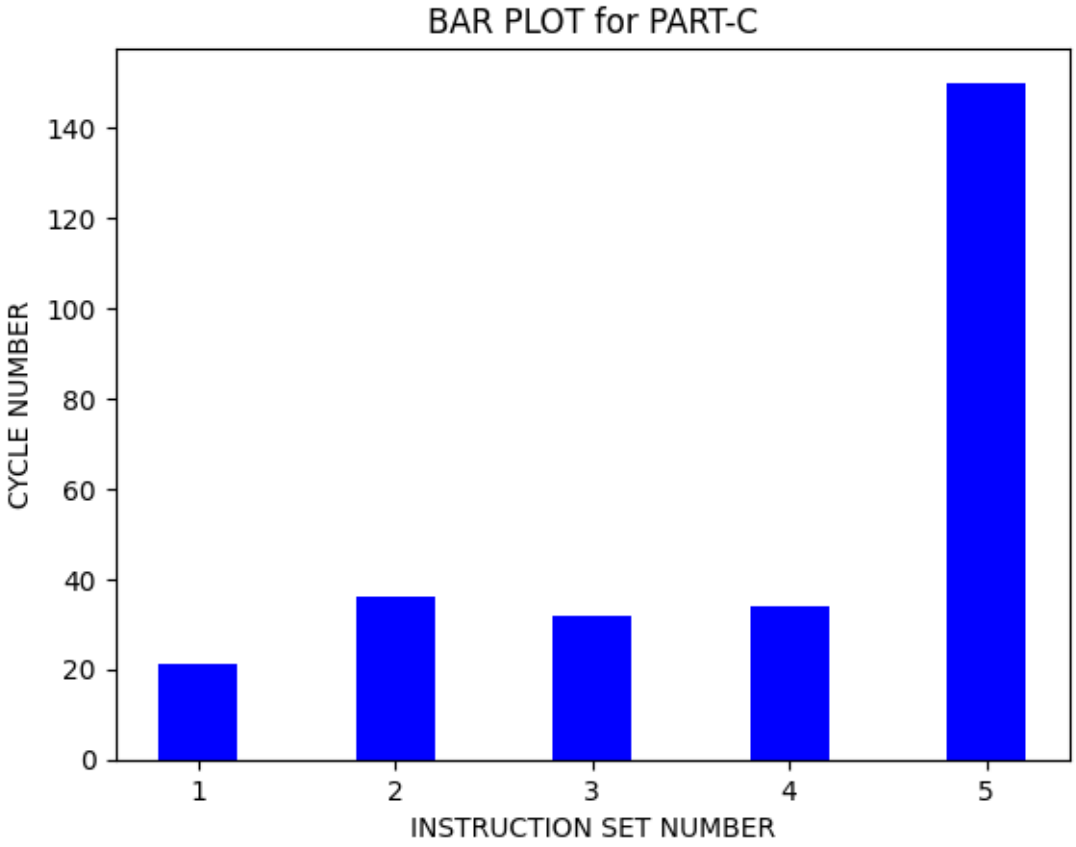



Figure 3: Part-3

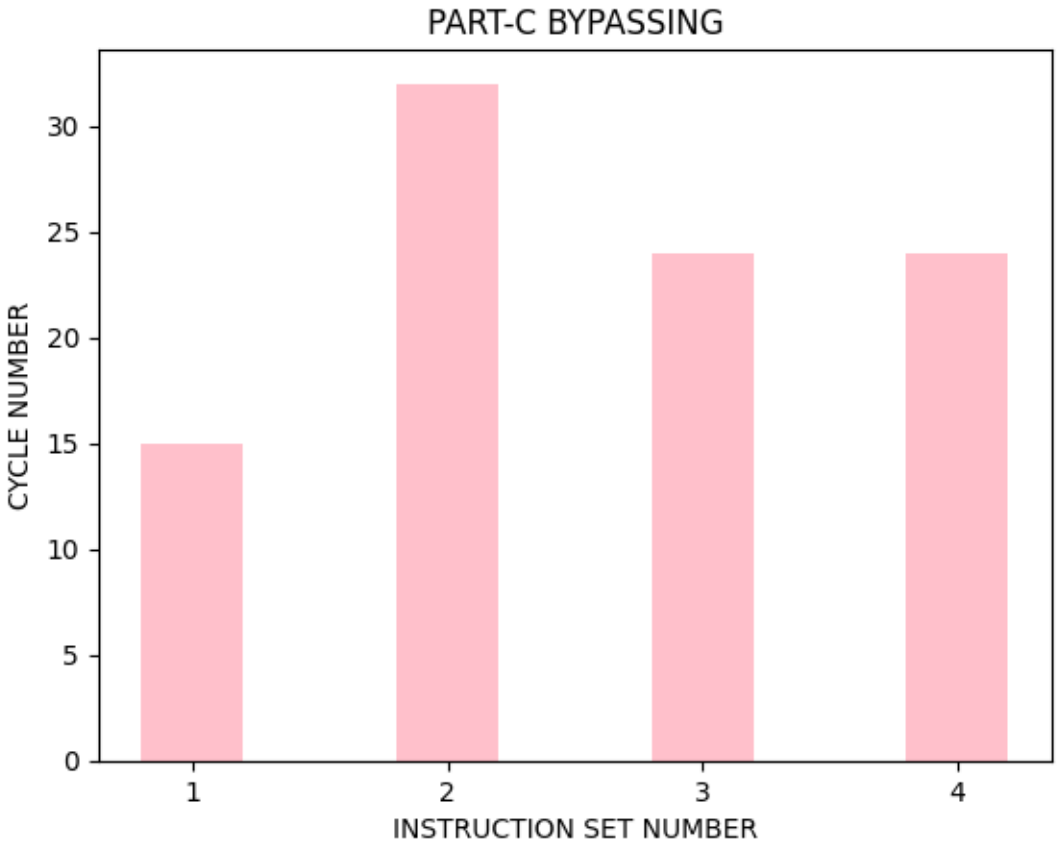



Figure 4: Part-3