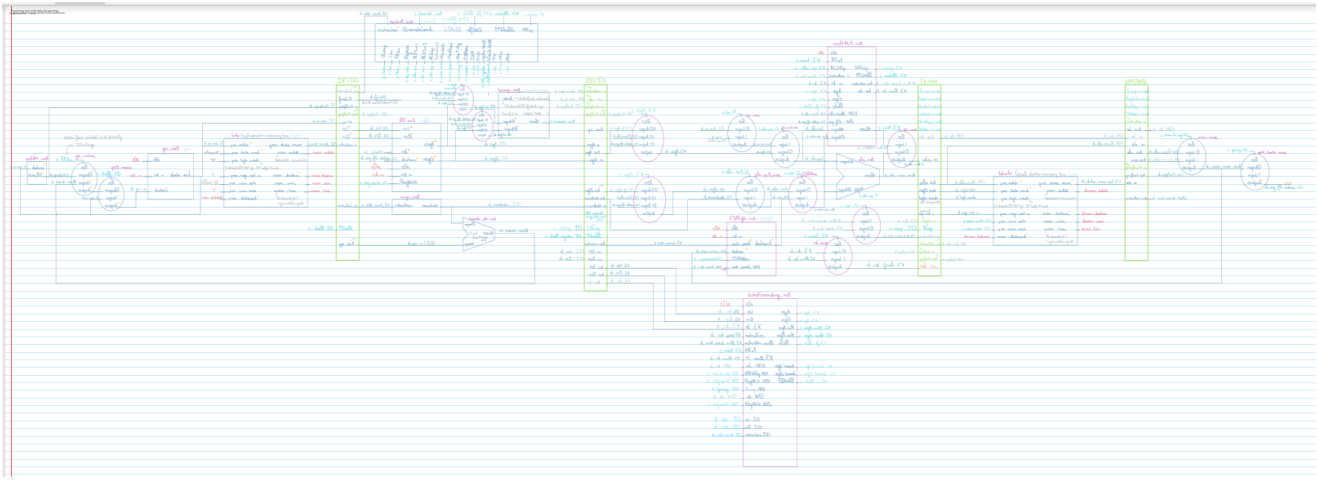
# Project Report - Part 2

## Datapath diagram

Below is a screenshot of the diagram



- Due to the fact that the figure is large, taking a screenshot of the full datapath will render the resolution really bad. I have provided a much clearer pdf of the datapath to consult in the deliverables.

## Pipelining

### New Components

- *branch_addr_alu* to calculate the address of the branch in ID stage
- *IF_ID* register bus
- *ID_EX* register bus
- *EX_MEM* register bus
- *MEM_WB* register bus
- *multALU* as the multiply unit
- *multALUbus* as the register bus inside the multALU unit pipelined

### Modifications in CSRfile

NB: This section can be skipped as it has no visible effects on the execution, it is just optimization

- The input instruction is used for both reading and writing to the CSRfile. But now each will be done in a different stage
- Reading is done in the EX stage
- Writing is done in the WB stage
- To solve this issue I added an additional instruction_WB input port. This will take the instruction of the WB register, and the initial instruction will therefore only be used for reading
- We also modified the instruction counter to read from the instruction_WB not EX
- The condition of instr_word_WB(0)=0 or 1 is used to make sure we don't count if the signal is undefined

### New Control signals

- Input:
  - MStall coming from multALU unit
  - MNop coming from multALU unit
- Output:
  - Stall: This is gonna be used for stalling in general. For our current purposes, MStall will tell us there is an instruction in the multiply unit. We check if the current instruction in ID is a multily instruction we let it pass so "Stall<=0", else we stall it so "Stall<=1"
  - Nop: This is gonna be used for sendind NOP to EX/MEM in general. For our current purposes, Nop<=MNop

- MExt: This is '1' when the instruction i have is a M extension instruction

## M extension instructions

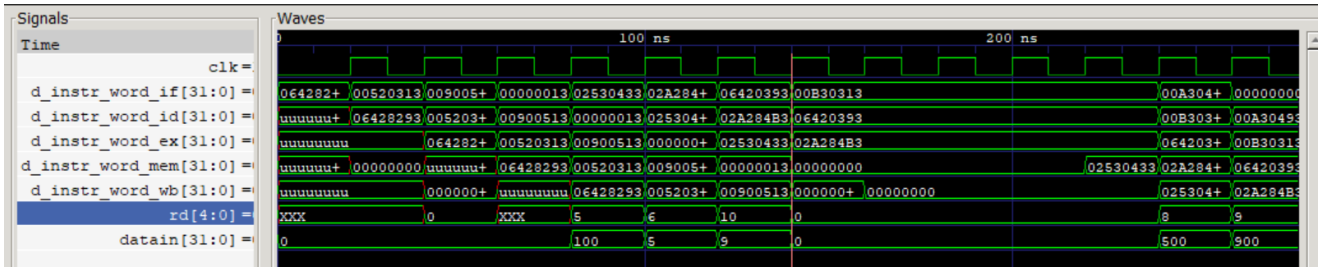
- He asked us to do a multiply unit that simulates 5 clock cycles to give a result
- I did a new multALUbus as well to simulate the pipelining inside the multiply unit alu
- When it comes to pipelining:
  - We made it in a way to support multiply pipelining
  - The only signals that we care to preserve from one multiply to the next are the result and the instruction. So we inputed the instruction in the mult unit, made it go throught the five stages and at the entrance of the EX_MEM bus there are 2 muxes that ensure we take the instruction and the result from the multiply unit
- When it comes to stalling:
  - MStall will go to 10 when MExt_EX1 becomes 1, if the instruction is a divide it will go to 11
  - MStall will go back to zero as soon as the instruction reaches the last execution stage (EX5) in the mult unit at the condition that the instruction in the EX1 and the instruction in EX5 are the same one to account for pipelined multiplication
- When it comes to NOP:
  - I only sent NOP to EX/MEM because the MEM/WB will inherit the NOP from it
  - As soon as the first multiply that entered the pipeline arrive to EX5, NOP turns back to 0

## Testing the multiply pipelined - test_04

The RISC-V code:

```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
00900513 addi x10 x0 9
00000013 nop
02530433 mul x8 x6 x5
02a284b3 mul x9 x5 x10
06420393 addi x7 x4 100
00b30313 addi x6 x6 11
00a30493 addi x9 x6 10
```

Please take the red line in the picture below as the cursor reference for the explanation below



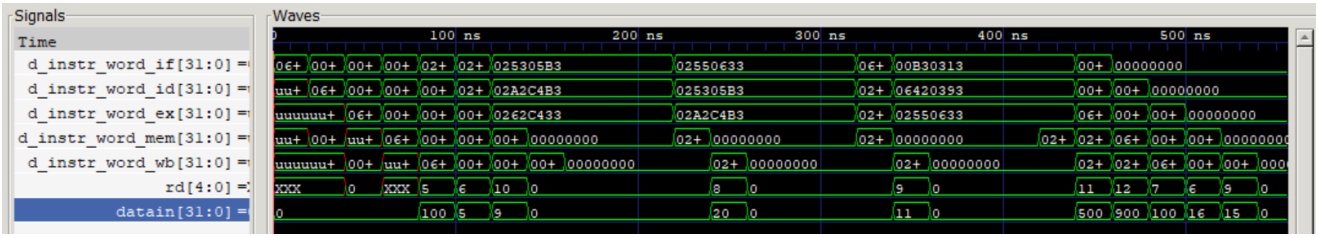

- We can first observe in the EX stage that the first multiply instuction enters but does not stall anything as the second multiply enters right after it
- After the second multiply we can observe everything being stalled for 5 cc.
- The addi that is after the second mult gets stalled in the ID and the addi that is after it gets stalled in the IF
- After the NOP passes through the mem stage, everything after it is just empty instructions (zeros) until the first multiply finishes executing, then it gets sent to mem and everyting continues normally
- Finally we can observe through rd and datain (these are the signals being sent to rf) that all instructions are being executed correctly, and all values are being written in the register file correctly

## Testing the divide pipelined - test_05

The RISC-V code:

```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
00900513 addi x10 x0 9
00000013 nop
0262c433 div x8 x5 x6
02a2c4b3 div x9 x5 x10
```

```
025305b3 mul x11 x6 x5
02550633 mul x12 x10 x5
06420393 addi x7 x4 100
00b30313 addi x6 x6 11
00a30493 addi x9 x6 10
```




- The clearest way to observe that it works is through rd and datain
- Other than that, reading on the ex instruction we can observe that the first and second divide stay for five cc each, and then the first multiply enters and directly after it the second multiply enters.
- The instructions also come out to the wb stage correctly, with full zeros whenever there is a stall in the ex

# Implementing Data Forwarding

## New Components and Elements modifications

- *dataForwarding* used as the data forwarding unit
- *bNop* signal for the ID/EX stage
- *bStall* and *dfStall* inputs for the control signal

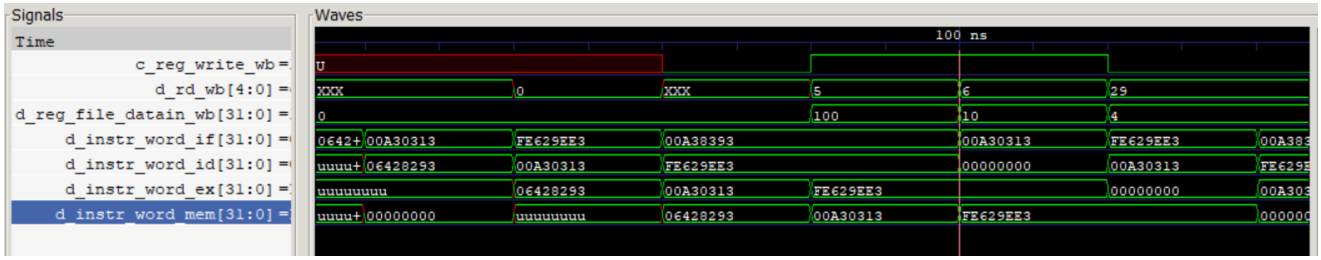
## Data Forwarding unit functioning

- For regular forwarding
  - It takes in rs1 and rs2 from the EX stage to compare them with rd from MEM and WB
  - rd coming from EX is used to make sure it is not zero, in case it does it would be useless to check for equality
  - The instruction coming from EX is used to check that rs1 and rs2 are used and aren't just buffer numbers (some instructions don't have rs1 and rs2)
  - RegWrite coming from MEM and WB are used to check that the instructions there will actually write back to there respective rd, oterwse the values coming from them are insignificant and shouldn't be forwarded
- For M extension forwarding
  - The concept goes mainly the same as for the normal forwarding, the difference is that when the mult(or div) is in the last EX stage inside the multiply unit, it will check for data forwarding there
- For branch forwarding
  - rs1 and rs2 are taken from ID to check for dependencies while calculating in the branch comparator
  - In case rs1 and rs2 are dependent on rd of EX we will assert bStall, which will case a stall in IF and ID and will send a NOP through the ID/EX bus
  - In case rs1 and rs2 depend on rd of MEM or WB, we will implement normal data forwarding
    → We had to create the *regularStall* signal for the ID/EX bus. In case a branch in ID depends on an instruction in EX, and this instruction in turn depends on a load in MEM, we need to give the priority for the stall coming from the EX instuction, and would want to avoid sending a nop in the bus until the next cc

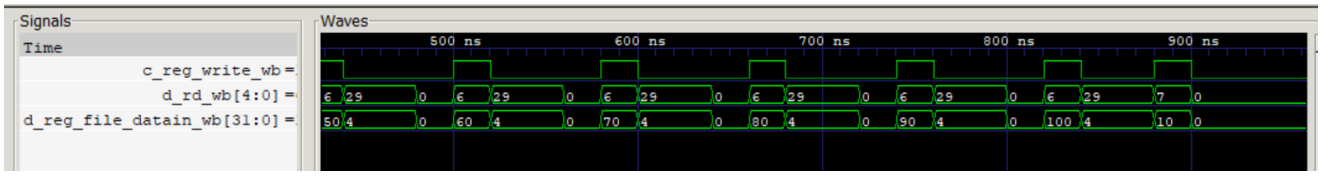
## Testing data forwarding for branch instructions - test_08

The RISC-V code:

```
06428293 addi x5 x5 100
                loop:
00A30313 addi x6 x6 10
FE629EE3 bne x5 x6 loop
00A38393 addi x7 x7 10
```




- In the above picture we can observe first how the instruction after the branch got fetched, but as soon the branch got resolved as taken, the fetched instruction got flushed and did not make it to the ID stage (right side of the cursor)

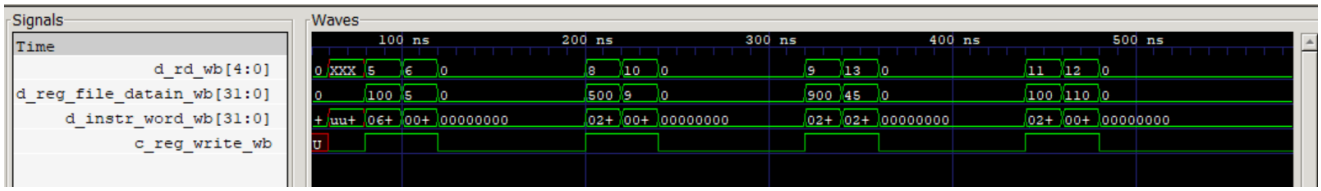



- We can first note that the instructions are being executed in a loop meaning the ranch is taken correctly
- Then we can notice that as soon as x6 becomes 100 no more instructions are taken meaning the branch got taken at the correct time
- And after it the last addi that comes right after the branch instruction got taken

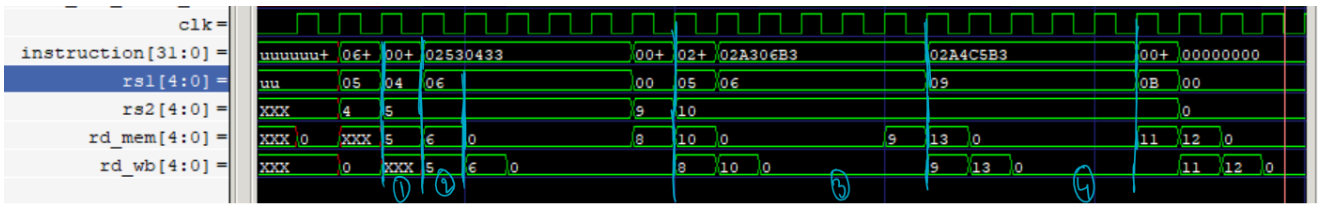
## Testing data forwarding with mult and div - test_07

The RISC-V code:

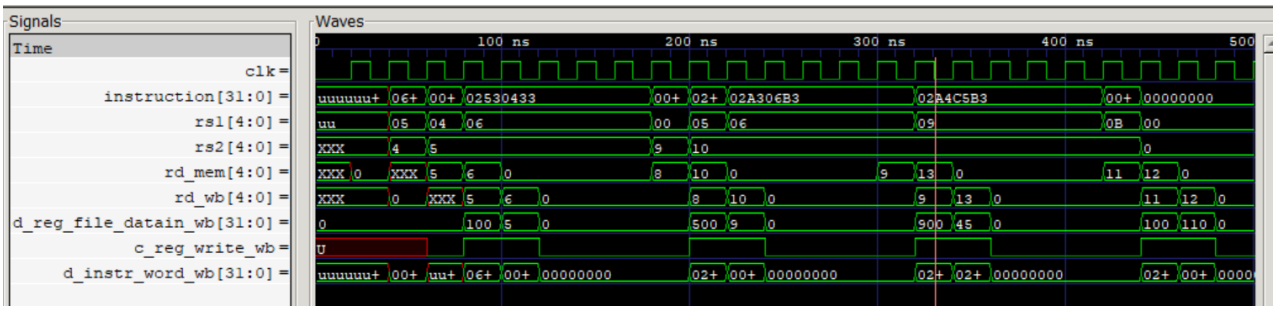
```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
02530433 mul x8 x6 x5
00900513 addi x10 x0 9
02A284B3 mul x9 x5 x10
02A306B3 mul x13 x6 x10
02A4C5B3 div x11 x9 x10
00A58613 addi x12 x11 10
```




- We can observe all the values being stored correctly in the correct registers.
- Refer to the below explanation for further detail




1. We can observe for the second addi that even though rs2 appears to be 5 and rd_mem is 5, it does not cause conflict because of the opcode condition stated, which takes into consideration that not all instructions have an rs2, some have immediate instead, even some instructions don't have rs1.
2. The first mul depends on both instructions before it. It takes regA from memory and regB from wb
3. Then we have 2 mul pipelined, they each successfully take values from wb and mem stages
4. We have the div whichwas stalled in the ID stage waiting for the mul to finish in the ex stage, then successfully took its regA value from the wb stage




# Implementing Hazard Detection

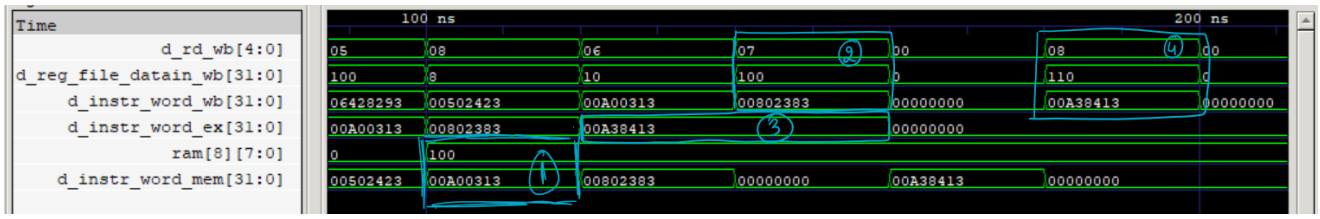
## Data forwarding with load

- I implemented this feature while working on the normal data forwarding as it is only a matter of checing the below
- MEMtoReg_MEM is used to check that the instruction in the memory stage isn't a load. In case it is the stall signal will be asserted
- Stall is sent to the control unit which will in turn assert the Stall signal coming out of it. It will stall the instructions in IF, ID and EX and will send a NOP to the MEM stage
- On the next clock cycle the instruction in the EX will be able to take the result of the load from the WB stage normally

## Testing data forwarding with load - test_06

The RISC-V code:

```
06428293 addi x5, x5, 100
00502423 sw x5, 8(x0)
00A00313 addi x6, x0, 10
00802383 lw x7, 8(x0)
00A38413 addi x8, x7, 10
```




1. We can see that the correct value was stored in ram at location 8 during the phase where sw instruction was in mem stage, so normal data forwarding works correctly
2. We can see how the value was fetched from memory and loaded into x7
3. We can see how the last addi was stalled in EX to wait for the load to arrive to the MEM stage and fetch the value
4. We can see that the correct value was calculated and stored in x8 for the last addi

# Handling Control Hazards

## Flushing

→ I need to flush the instruction fetched after a taken branch or a jump instruction

- I can do this relying on the c_PCSrc, if it is '1' it means I want to choose the addres coming from the branch_addr alu so it means I shoudn't have taken the pcplus4 hence I need to flush.
- To flush all i need to do is just inside the *IF_ID* register I put an if condition and set every output port to 0 in case c_PCSrc ='1'

→ Testing for flushing was show earlier in the data forwarding section (test_08)

# Each team member's unique contributions to the project

The coding part was done in common, we made a github repository to keep track of the code somewhere we can both access it but no significant changes were made individually. We would meet and progress in the work together.
As for te report, Geo provided the testing and Christie elaborated on the changes in the code and their functionality along with the drawing of the datapath.