In the assignment 2, I design the pipeline RISC-V in 4 aspects.

1. Five Stages
   Design five stages is the major assignment of pipeline RISC-V. And I implements it in the build-in structures provided by professor Miller: fetch, decode, execute, memory and writeback.
   a. Fetch: In this step, the main problem is how to fetch the provided program counter, and get the instruction from the program counter.
   b. Decode: Get the instruction, then use the instruction to get value of imm and resource registers, and pass them to execution stage.
   c. Execution: ALU. Calculate the resource registers and imms, and get corresponding values, which may belong to address or destination register. Then pass these values to memory stage.
   d. Memory: If the value passed from execution stage is about memory read or memory write, implement the read memory and write memory function here. If the value passed from execution stage is about destination register, pass it to writeback stage directly.
   e. Writeback: write the value from memory stage to the destination register.
2. Forward
   a. MX here

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **addi x1, x0, 1** | **F** | **D** | **X** | **M** | **W** | | |
| **addi x2, x1, 1** | | **F** | **D** | **X** | **M** | **W** | |

In the fourth cycle, MX forward will happen. If the execution stage finds one of the resources register have same address with the destination register of previous instruction in memory stage, get the value from the destination register and let rs = rd.

   b. WM here

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **lb x1, 0(x3)** | **F** | **D** | **X** | **M** | **W** | | |
| **sb x1, 0(x10)** | | **F** | **D** | **X** | **M** | **W** | |

In the fifth cycle, WM forward will happen. If the memory stage finds the resources register have same address with the destination register of previous instruction in writeback stage, get the value from the destination register and let rs =rd.

   c. WX
   In the fifth cycle, WX forward will happen. But I do not need to implement it in this assignment. Because W stage is before X and I have already write the value, which may cause the data hazard to the execution stage.

3. Stall

Load

Find at here

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **lb x1, 0(x3)** | **F** | **D** | **X** | **M** | **W** | | | |
| **sb x2, 0(x1)** | | **F** | **D** | **D** | **X** | **M** | **W** | |
| | | | **F** | **F** | **D** | **X** | **M** | **W** |

Find at here

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **lb x1, 0(x3)** | **F** | **D** | **X** | **M** | **W** | | | |
| **addi x2, x1, 1** | | **F** | **D** | **D** | **X** | **M** | **W** | |
| **addi x2, x2, 1** | | | **F** | **F** | **D** | **X** | **M** | **W** |

To implement stall, I set a boolean in stage_reg_e. The stall is found in execution stage. And only load instructions could lead to stall. When execution stage finds stall, I can just let the boolean in stage_reg_e be true.

And in the decode stage of second instruction, I set a if condition to check if the boolean is true. If in decode stage the boolean from stage_reg_e is true, the *new_x_reg will get all elements as 0, and this *new_x_reg will be passed to the execution stage of second instruction. But because every elements in the *new_x_reg is zero, nothing will happen in the fourth circle in execution stage. So the stall is implemented in the second instruction.

In the third instruction, I just copy the current_stage_d_register to *new_d_register and set the program counter stay at the same position. Therefore the stall is implemented in third instruction.

Then in the fourth cycle, when I launch the memory stage of the first instruction, I set the boolean in stage_reg_e be false. Then everything start to work in the normal way.

The decode stage of second instruction checks that the boolean become false, then pass corresponding values to the execution stage in second instruction.

The fetch stage of third instruction checked the boolean become false, then pass the new instruction to decode stage in third stage, and let set_pc(get_pc() +4).

4. Branch

I initial an array called BTB in the global filed.

BTB table (example instruction: 0b0000 0|000 10|00 beq x0, x0, L1)

tag index instruction

| **index** | **tag** | **Target address** | **status** |
|---|---|---|---|
| **00010** | **00000** | **L1 (pc+imm)** | **1** |

| | | | |
|---|---|---|---|
| **…...** | **…..** | **…..** | **…..** |

In the BTB table, I have 32 entry, so I need 2^5 bits for index. Because program counter is a multiplier of 4, so the least significant two bits are useless. Then the 3-7 bits are the index of the BTB table. The tags are used to confirm the entry. Target address is calculated based on the RISC-V manual. The status is for 2 bits prediction, but because I did not implement 2 bits prediction it, the status attribute always keeps constant 1.

When forward branch happen, because the program counter after the branch is greater than the previous program counter, nothing will happen.

When backward branch happen, because the program counter after the branch is less than the previous program counter, the index, tag, target address will be record on the BTB.

When the program counter in fetch stage match one entry in the BTB table, set_pc(target address).

About flush, when set a new program counter in memory stage, just send a stage_reg_d or stage_reg_x or stage_reg_m or stage_reg_w, which has a branchBoolean as 1, to next stage in second and third instruction. In my stage function(), I set a if condition: if a current_stage_dxmw_register has a branchBoolean as 1, then pass a stage struct has a branchBoolean as 1 too. In that way, when a branch happen after memory stage in first instruction, the execution stage of second instruction and decode stage of third instruction will know they will be flushed, and the rest of second instruction and rest of third instruction will also know they will be flushed continuously.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **F** | **D** | **X** | **M** | **W** | | | |
| | **F** | **D** | ~~**X**~~ | ~~**M**~~ | ~~**W**~~ | | |
| | | **F** | ~~**D**~~ | ~~**X**~~ | ~~**M**~~ | ~~**W**~~ | |
| | | | **F** | **D** | **X** | **M** | **W** |

BUG: mult